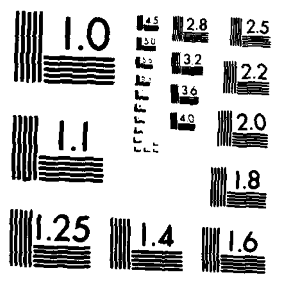
AD-A086 690 CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB F/6 9/2
THE DESIGN METHODOLOGY OF DISTRIBUTED COMPUTER SYSTEMS.(U)
MAY 80 C V RAMAMOORTHY AFOSR-78-3630
UNCLASSIFIED AFOSR-TR-80-0542 NL

1 OF 2
AD
A086690

1 OF 2

AD A086690



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS [illegible]

FINAL REPORT

THE DESIGN METHODOLOGY OF DISTRIBUTED COMPUTER SYSTEMS

by

C. V. Ramamoorthy

Final Technical Report

July 1, 1978 - June 30, 1979

GRANT AFOSR-78-3630

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data E

19 REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING ~~FORM~~
18 1. REPORT NUMBER AFOSR-TR-80-0542 / 2. GOVT ACCESSION NO. AD-A086690	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 THE DESIGN METHODOLOGY OF DISTRIBUTED COMPUTER SYSTEMS	9 5. TYPE OF REPORT & PERIOD COVERED Final technical rept. 1 Jul 77-30 Jun 79
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 C. V. Ramamoorthy	15 ~~8. CONTRACT OR GRANT NUMBER(s)~~ AFOSR-78-3630
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of California, Berkeley Electronics Research Laboratory Berkeley, CA 94720	10. PROGRAM ELEMENT, PROJECT, TASK ~~AREA & WORK UNIT~~ NUMBERS 16 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB, Washington, DC 20332	11 ~~12. REPORT DATE~~ May 1980 17 A2
	~~13. NUMBER OF PAGES~~ 99
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

12 101

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Petri net
concurrent system deadlock
binary semaphore
critical region
adaptive reconfiguration
reconfiguration strategies
figure of merit

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Performance evaluation techniques for asynchronous concurrent systems is developed using the Petri net approach. Analysis techniques for deadlocks in asynchronous concurrent systems is also explored. Moreover, a need for adaptive reconfiguration techniques is established along with necessary and sufficient conditions for reconfigurability. Then a method is described for evaluating the available reconfiguration strategies based on a figure of merit is described.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE UNCLASSIFIED 127550 JM ntered)

~~IGE~~

THE DESIGN METHODOLOGY OF DISTRIBUTED COMPUTER SYSTEMS

by

C. V. Ramamoorthy

Final Technical Report

July 1, 1978 - June 30, 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of Califronia, Berkeley
94720

1. Introduction

In the past year, we have studied the design methodology for distributed computer systems. We have developed analysis techniques to facilitate the design of distributed computer systems in general, and developed a design methodology as well as modelling techniques for fault tolerant distributed computer systems in particular. The work that we have accomplished can be summarized as follows: (1) performance evaluation of asynchronous concurrent systems (2) methods to detect deadlock in distributed systems (3) design methodology and modelling techniques for fault-tolerant computer systems.

The techniques for prediction and verification of the performance of asynchronous concurrent systems can be classified into two categories (1) deterministic models, and (2) probabilistic models. In deterministic models, it is usually assumed that the task arrival times, the task execution times, and the synchronization involved are known in advance to the analysis. With this information, a very precise prediction of the system performance can be obtained. This approach is very useful for performance evaluation of real time control systems with hard deadline requirements.

In probabilistic models, the task arrival rates and the task service times are usually specified by probabilistic distribution functions. The synchronization among tasks is usually not modelled, because otherwise the number of system states becomes so large that it would be impossible to perform any analyses. Probabilistic models usually give a gross prediction on the performance of the system and are good for early stages of system design when the system characteristics are not well understood. In this paper, we focus on performance analysis of real time systems and therefore we have chosen the deterministic approach. In particular, in order to model clearly the synchronization involved in concurrent systems, the Petri net model is chosen.

In our approach, the system to be studied is first modelled by a Petri net. Based on the Petri net model, a given system is classified as either (Fig. 1.1): (1) a consistent system; or (2) an inconsistent system (the definitions are given in later sections of the paper). Most real-world systems fall into the first class and so we focus out discussion on consistent systems. Due to the difference in complexity involved in the performance analyses of different types of consistent systems, they are further subclassified into: (i) decision-free systems; (ii) safe persistent systems; and (iii) general systems. Procedures for predicting and verifying the system performance of all three types are presented. It is found that the computational complexity involved increases in the same order as they are listed above.

Our work in system deadlocks concentrates on analysis techniques for deadlocks in asynchronous concurrent systems. This includes multi-programmed systems, multiple processor systems and computer networks. In particular, we study in detail deadlocks caused by conflicts in mutual exclusive accesses to resources with the constraint that each resource type has only one member. Deadlocks due to the erroneous nesting of binary semaphores [Dij 71], nesting of critical regions [Bri 72, Bri 73a] and nesting of monitors [Bri 73b, Hoa 74] are important members in the above category. In addition to these, deadlocks due to conflicts in data file lockings in distributed database systems also fall into the above category.

Our work on fault tolerant distributed computer systems encompasses the issues of establishing a systematic design procedure, development of models and analysis techniques to analyze the recovery process and to provide generalized design techniques. In striving for this goal there are several issues that have to be faced. First one is the lack of any established definition and classification of distributed systems. Secondly, there is no established design methodology following which, on could design reconfigurable distributed systems. Thirdly, there are no modelling and analysis techniques which are directly applicable to model and analyze the distributed fault tolerance. In order to overcome these difficulties, we have studied the following problems: First, a generalized scheme for failure classification is proposed and several failures occuring in distributed systems are identified and classified according to the proposed classification. A need for adaptive techniques is established and various methods of reconfiguration are discussed. Based on our model a set of necessary and sufficient conditions are developed for reconfigurability of a system. A systematic procedure of expressing a given situation in terms of the model components is given to test the validity of a system design. Basically, in this approach, a failure is considered as an "observation event". A set of fault equations are developed in terms of a set of basic events and failure transfer functions. For a given failure a mincut set derived from the analysis would indicate whether reconfiguration is feasible.

The next aspect of our reconfiguration study is to evaluate a set of possible reconfiguration strategies and to select the one that is best suited for a given dynamic environment. The evaluation is based on the type and nature of faults, performance degradation and other real time constraints. In order to make a quantitative analysis to compare and to select various reconfiguration strategies, a figure of merit (FOM_R) is defined. This FOM_R is defined in terms of a vitality factor (V_R), feasibility factor (F_R), reliability of that selected strategy in terms of the reliability of the participating components (R_R), and a normalized cost factor (C_{ncf}). Second, a model, based on the graph theory is proposed that would allow the representation of various attributes, like reliability, distributed control etc for analysis. This Unified Graph Model (UGM) represents the actions or computations as a set of nodes of a graph in which arcs connecting the nodes represent the flow of status. Any entity to be analysed such as reliability of a path, or a subsystem or a reconfiguration strategy or the consistency of a communication protocol etc., is expressed in terms of this model. Thirdly, the issue of adaptive reconfiguration is studied. Specifically, it addresses two fundamental questions: (1) feasibility of reconfiguation, 2) synthesis and selection of reconfiguration strategy. The proposed model enables the designer to analyze whether a given design would be able to recuperate from the failures and also identifies the weakest spots in the design. Lastly, we demonstrates the applicability of the proposed modelling and analysis approaches, and presents an overview of a design for a candidate distributed system based on the concepts developed in earlier chapters. The requirements for this example system are first established and the architecture is presented. The proposed methodology, modelling and analysis techniques are applied.

This report is divided into 8 section. Section 2 presents the techniques for performance evaluation in concurrent system. Section 3 develops the procedure of deadlock detection in distributed system. Section 4 concentrates on the reliability issue of distributed systems. Section 5 deals with the issue of design methodology related to the design of reconfigurable systems. Specifically, a design methodology is proposed and summarized. Section 6 presents a detailed discussion on the study of adaptive reconfiguration. Section 7 gives an example system to demonstrate the applicability of our results. Lastly, section 8 gives a conclusion of the report.

2. Performance Evaluation of Asynchronous Concurrent System

2.1 Review on Petri Nets

2.1.1 Basic Properties of Petri Nets

Petri nets [PET 77, AGE 75] are a formal graph model for modelling the flow of information and control in systems, especially those which exhibit asynchronous and concurrent properties. A Petri net contains two types of nodes: the circles (called places) represent conditions and the bars (called a token) at a place indicates the holding of the condition of the place. A pattern of tokens in a Petri net (called a marking) represents the state of the system.

To model the dynamic behavior of a system, the execution of a process is represented by the firing of the corresponding transition. The changes in system state are represented by the movements of tokens in the net. The firing rules of Petri nets are:

(1) A transition is enabled if and only if each of its input places has at least one token.

(2) A transition can fire only if it is enabled.

(3) When a transition fires:

 (i) a token is removed from each of its input places; and

 (ii) a token is deposited into each of its output places.

2.1.2 Application of Petri Nets in Control Flow Analysis

Petri nets have been used extensively to study the control flow of computer systems. By analyzing the liveness, boundedness and proper termination properties of the Petri net model of a computer system, many desirable properties of the system can be unveiled.

A Petri net is live [HAC 75, HOL 71] if there always exists a firing sequence to fire each transition in the net. By proving that the Petri net is live, the system is guaranteed to be deadlock free.

A Petri net is bounded [KAR 66, LIE 76] if for each place in the net, there exists an upper bound to the number of tokens that can be there simultaneously. If tokens are used to represent intermediate results generated in a system, by proving that the Petri net model of the system is bounded, the amount of buffer space required between asynchronous processes can be determined and therefore information loss due to buffer overflow can be avoided. If the upper bound on the number of tokens at each place is one, then the Petri net is safe. Programming constructs like critical regions [BRI 72] and monitors [BRI 73, HOA 74] can be modelled by safe Petri nets.

A Petri net is properly terminating [GOS 71, POS 74] if the Petri net always terminate in a well-defined manner such that no tokens are left in the net. By verifying that the Petri net is properly terminated, the system is guaranteed to function in a well behaved manner without any side-effects on the next initiation.

2.1.3 Extended Timed Petri Nets

In order to study the performance of a system, the Petri net model is extended to include the notion of time [RAM 74]. In such extended nets, an execution time, τ, is associated with each transition. When a transition initiates its execution it takes τ units of time to complete its execution. With the extended Petri net model the performance of a computer system can be studied.

2.2 Performance Evaluation

The work that we have accomplished in performance evaluation is to use Petri nets to find the maximum performance of the system, i.e., to find the minimum cycle time (for processing a task) of the system. As pointed out before, different computational complexities are involved in the analyses of systems of different types. The approaches for analyzing each type of system are studied separately in detail in the following section. Before we come to the analyses, some definitions are in order.

Definition. In a Petri net, a sequence of places and transitions, $P_1t_1P_2t_2...P_n$, is a directed path from P_1 to P_n if transition t_i is both an output transition of place P_i and an input transition of place P_{i+1} for $1 \leq i \leq n-1$.

Definition. In a Petri net, a sequence of places and transitions, $P_1t_1P_2t_2...P_n$, is a directed circuit if $P_1t_1P_2t_2...P_n$ is a directed path from P_1 to P_n and P_1 equals P_n.

Definition. A Petri net is strongly connected if every pair of places is contained in a directed circuit.

In this report, we presented the performance analysis techniques for stongly connected non-terminating Petri nets. Extensions to analyze weakly connected Petri nets are quite straightforward so it will not be discussed in this report.

2.2.1 Consistent and Inconsistent Systems

The first step involved in our approach to analyze the performance of a system is to model it by a Petri net. A system is a consistent (inconsistent) system if its Petri net model is consistent (inconsistent). A Petri net is consistent (condition A) if and only if there exists a non-zero integer assignment to its transitions such that at every place, the sum of integers assigned to its input transitions equals the sum of integers assigned to its output transitions; otherwise, the system is inconsistent. If a transition has n input arcs to a place, it is counted as n input transitions to that place.

Figure 2.1a is an inconsistent system and Fig. 2.1b is a consistent system. In Fig. 2.1a, there does not exist an integer assignment to its transitions to satisfy condition A. This can be verified by assigning an integer variable to each transition and getting a contradiction in trying to solve the simultaneous equations provided by condition A:

For place A: $x + y = z$ (i)

For place B: $x = z$ (ii)

For place C: $y = z$ (iii)

(ii) + (iii): $x + y = 2z$ (iv)

and therefore eq. (iv) contradicts eq. (i).

Figure 2.1b is a consistent Petri net. If each transition is assigned an integer of value 1, condition A is satisfied.

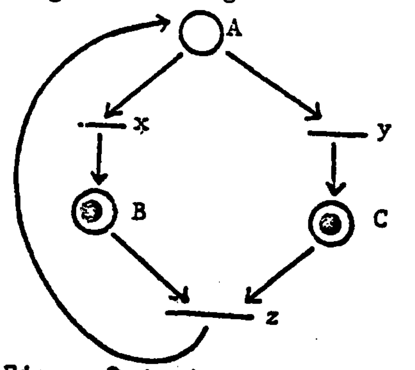


Figure 2.1a An inconsistent system

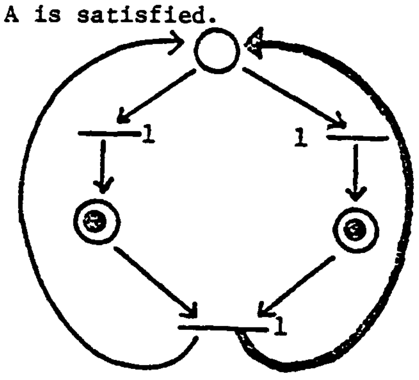


Figure 2.1b A consistent system

The practical implication behind this system classification is that the integer assigned to a transition is the relative number of executions of that transition in a cycle. If a system is live and consistent, the system goes back to its initial configuration (state) after each cycle and then repeats itself. If a system is inconsistent, either it produces an infinite number of tokens (i.e., it needs infinite resources) or consumes tokens and eventually comes to a stop. Most real-world systems which function continuously with finite amount of resources fall into the class of consistent systems, hence, we focused our discussion on consistent systems and further sub-classified them into decision-free systems, persistent systems and general systems. Performance analysis techniques for each subclass are discussed in the following subsections.

2.2.2 Decision-free Systems

A system is a decision-free system if its Petri net model is a decision-free Petri net (also known as marked graph [COM 71, MUR 77]). A Petri net is decision-free if and only if for each place in the net, there is one input arc and one output arc. This means that tokens at a given place are generated by a predefined transition (its only input transition) and consumed by a predefined transition (its only output transition).

The computer configuration shown in Fig. 2.2 is a decision-free system. The train system shown in Fig. 2.3 is another decision-free system. The tokens in the net are used to represent trains. For the convenience of the passengers, trains wait at stations for the next train to arrive so as to allow passengers to transfer between trains before leaving stations. Similarly, the chaining operations in the CRAY-1 computer RUS 78 can b modeled by a decision-free Petri net such as Fig. 2.3. The results issued from one function unit are immediately fed into another functional unit and so on. For a decision-free system, the maximum performance can be computed quite easily. However, before we come to that result, we need the following two theorems.

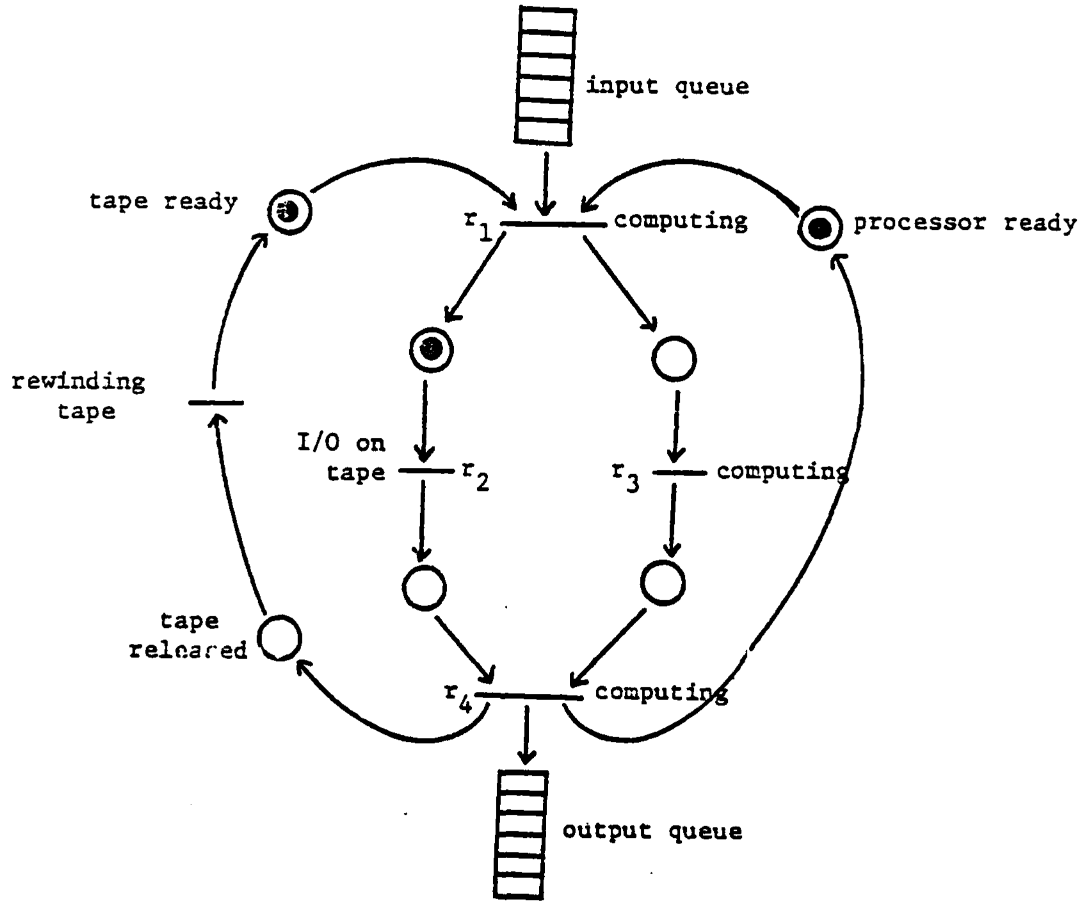


Figure 2.2 The Petri net model of a computer configuration

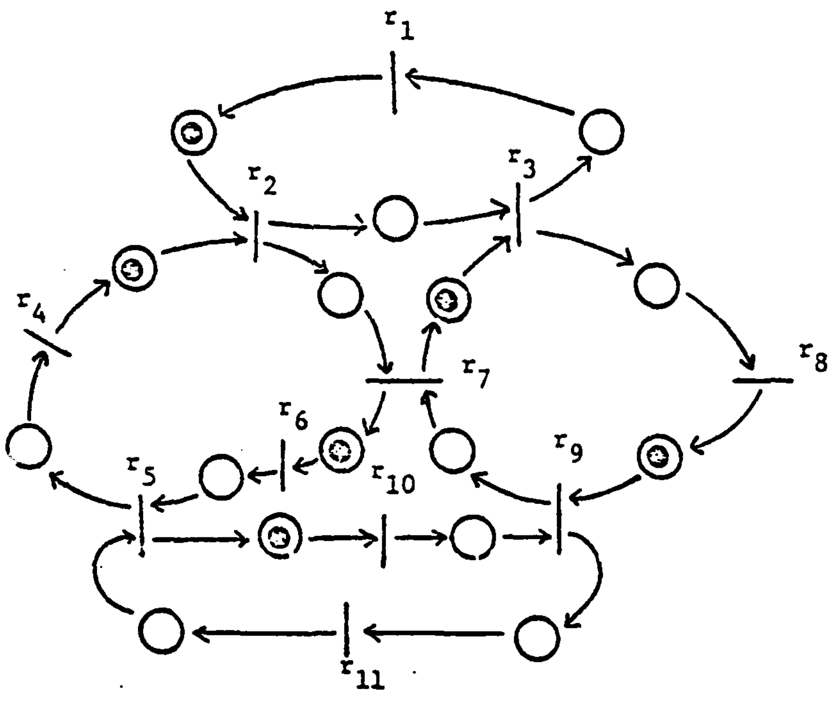


Figure 2.3 The Petri net model of a train configuration

Theorem 2.1. For a decision-free Petri net, the number of tokens in a circuit remains the same after any firing sequence. This result has been proven by many researches [COM 71, MVR 77], so the proof is not included in this report.

Definition. Let $S_i(n_i)$ be the time at which transition t_i initiates its n_i-th execution. The cycle time, C_i, of transition t_i is defined as

$$\lim_{n_i \to \infty} S_i(n_i)/n_i .$$

Theorem 2.2. All transitions in a decision-free Petri net have the same cycle time.

Proof: Consider transitions t_i and t_j in a decision-free Petri net. Choose a circuit that contains both transitions t_i and t_j. (Such a circuit must exist because the net is strongly connected.) Without loss in generality, assume that initially there are M_a, M_b, M_c, M_d and M_e tokens in the places in the chosen circuit as shown in Fig. 2.4. At time $S_i(n_i)$,

$$\begin{aligned} n_i - M_e - M_d &\leq \# \text{ initiations of transition } t_j \\ &\leq n_i + M_a + M_b + M_c \end{aligned}$$

$$\lim_{n_i \to \infty} \frac{S_i(n_i)}{n_i - M_e - M_d} \geq C_j \geq \lim_{n_i \to \infty} \frac{S_i(n_i)}{n_i + M_a + M_b + M_c}$$

Since M_a, M_b, M_c, M_d and M_e are finite, as $n_i \to \infty$, the left and right hand side expressions approach C_i, i.e.,

$$C_i \geq C_j \geq C_i, \quad C_i = C_j$$

Therefore, all transitions in a decision-free Petri net have the same cycle time, C.

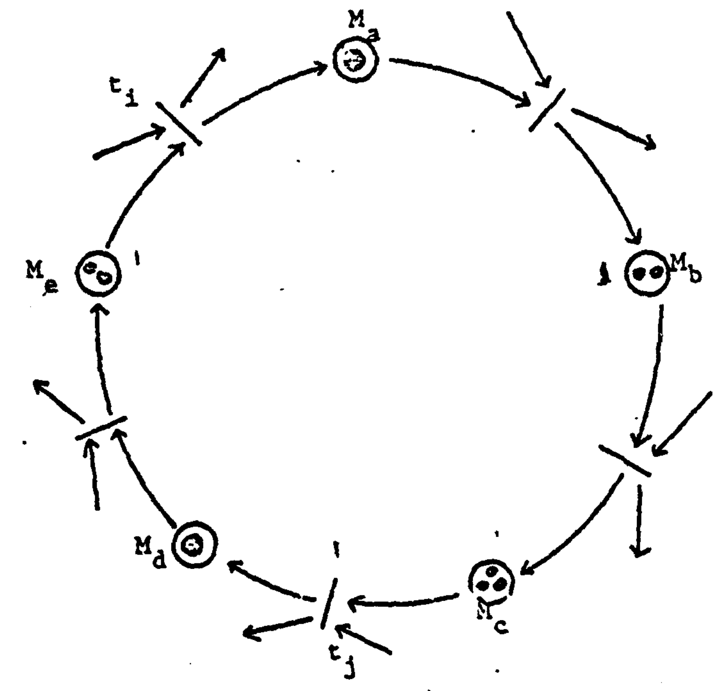


Figure 2.4 A cycle in a decision-free Petri net

Theorem 2.3. For a decision-free Petri net, the minimum cycle time (maximum performance) C is given by

$$C = \max\left\{\frac{T_k}{N_k} : k = 1,2,\ldots,q\right\}$$

such that $S_i(n_i) = a_i + Cn_i$

$T_k = \sum_{t_i \in L_k} r_i$ = sum of the execution times of the transitions in circuit k

$N_k = \sum_{P_i \in L_k} M_i$ = total number of tokens in the places in circuit k

q = number of circuits in the net

a_i = constant associated with transition t_i

L_k = loop (circuit) k

M_i = number of tokens in place P_i

Similar results have been obtained in [REI 68, MVR 77]. The proof given here uses a graph theoretical appraoch and is different from the previous approaches. Based on this approach, we develop a very fast procedure to verify the performance of a system.

Proof of Theorem 2.3: The proof is in two parts:

(A) Minimum cycle time, $C \geq \max\left\{\frac{T_k}{N_k} : k=1,2,\ldots,q\right\}$

(B) For $C = \max\left\{\frac{T_k}{N_k} : k=1,2,\ldots,q\right\}$ there exists a_i, such that $S_i(n_i) = a_i + Cn_i$ and the firing rules are not violated.

Proof of A:

# of transitions that are enabled simultaneously $\leq$ \# of tokens in circuit $= N_k$ (Theorem 2.1) processing power required by circuit per cycle $= T_k = \sum_{t_i \in L_k} r_i$

$\leq$ maximum processing power of the circuit per cycle time

$= CN_k$

therefore $T_k \geq CN_k$ for every circuit and $C \geq \max\{T_k/N_k, k=1,2,\ldots,q\}$.

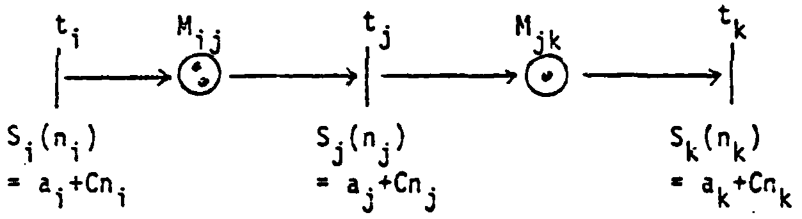
<u>Lemma</u>. For $C = \max\{T_k/N_k,\ k = 1,2,\ldots,q\}$,

$0 \geq T_k - CN_k$ for all circuits k.

<u>Proof</u>. $C \geq T_k/N_k,\ k = 1,2,\ldots,q$

$0 \geq T_k - CN_k$.

<u>Proof of (B)</u>: Let $C = \max\{T_k/N_k,\ k = 1,2,\ldots,q\}$.



In order not to violate the firing rules.

Finish time of the n_i-th execution of transition t_i

$\leq$ Initiation time of the n_i+M_{ij} execution of transition t_j

$$S_i(n_i) + r_i \leq S_j(n_i+M_{ij})$$

$$a_i + Cn_i + r_i \leq a_j + C(n_i+M_{ij})$$

$$a_i - CM_{ij} + r_i \leq a_j \qquad \text{(i)}$$

Similarly $a_j - CM_{jk} + r_j \leq a_k$ (ii)

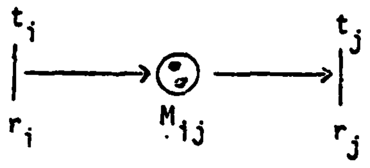
(i) + (ii) $a_i - C(M_{ij}+M_{jk}) + r_i + r_j \leq a_k$

In general $a_i - C\sum_{(u,v)\in R} M_{uv} + \sum_{w\in R} r_w \leq a_s$ (iii) where R is a path from transition i to transition s. In order not to violate the firing rules, we have to find a_i's such that (iii) is satisfied.

Procedure for assigning a_i's such that

$$a_i - C \sum_{(u,v) \in R} M_{uv} + \sum_{w \in R} r_w \leq a_s \qquad \text{(iii)}$$

(1) Define the distance from transition t_i to transition t_j (t_i adjacent to t_j) to be $r_i - CM_{ij}$.



(2) Find a transition t_s, which is enabled initially and assign 0 to a_s.

(3) Assign a_u to each transition, t_u, such that a_u is the greatest distance from t_s to t_u.

i.e. $$a_u = \max\left\{\sum_{w \in R} r_w - C \sum_{(u,v) \in R} M_{uv}\right\}$$

where R is a path from t_s to t_u.

Such an assignment of a_i's exists because by the Lemma, $T_k - CN_k \leq 0$, the greatest distance between any two nodes is finite and the corresponding path would never contain a loop. Q.E.D.

A drawback of the above approach is that all circuits in the net must be enumerated; this can be very tedious. In the design of computer systems, the required performance is usually given. With this information, the performance of a system can be verified very efficiently. By the Lemma, the performance requirement (expressed in cycle time, C) can be satisfied if and only if $CN_k - T_k \geq 0$ for all circuits. This can be verified by the following procedure.

A procedure for verifying system performance

(1) Express the token loading in an nxm matrix, P, where n is the number of places in the Petri net model of the system. Entry (A,B) in the matrix equals x if there are x tokens in place A, and place A is connected directly to place B by a transition. Matrix P of the example system in Fig. 2.5 is shown below:

$$\begin{array}{c|ccccccc|}
 & A & B & C & D & E & F & G \\
A & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
B & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
C & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
D & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
E & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
F & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
G & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Matrix P

(2) Express transition time in an nxm matrix, Q. Entry (A,B) in the matrix equals to r_i (execution time of transition i) if A is an input place of transition i and B is one of its output places. Entry (A,B) contains the symbol "w" if A and B are not connected directly as described above. Matrix Q for the example system is:

$$\begin{array}{c|ccccccc|}
 & A & B & C & D & E & F & G \\
A & w & w & 5 & 5 & w & w & w \\
B & w & w & 5 & 5 & w & w & w \\
C & w & w & w & w & 20 & w & w \\
D & w & w & w & w & w & 4 & w \\
E & w & 3 & w & w & w & w & 3 \\
F & w & 3 & w & w & w & w & 3 \\
G & 2 & w & w & w & w & w & w
\end{array}$$

Matrix Q

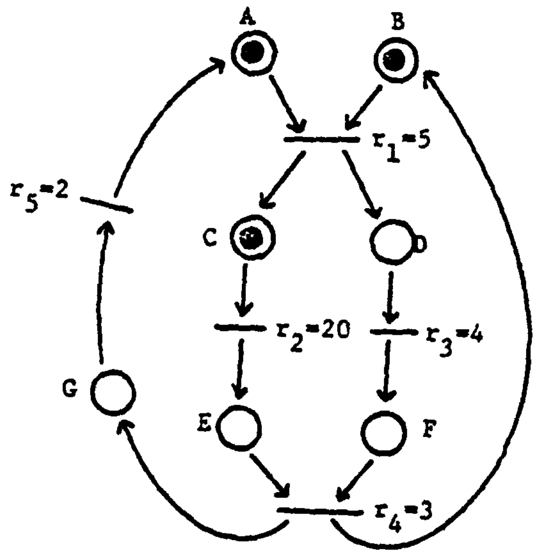


Figure 2.5 A computer configuration with the execution times of its processes

(3) Compute matrix CP-Q (with n-w = oo for $n \in N$), then use Floyd's algorithm [Flo 62] to compute the shortest distance between every pair of nodes using matrix CP-Q as the distance matrix. The result is stored in matrix S. There are three cases:

(a) All diagonal entries of matrix S are positive (i.e., $CN_k - T_k > 0$ for all circuits) -- the system performance is higher than the given requirement.

(b) Some diagonal entries of matrix S are zero's and the rest are positive (i.e., $CN_k - T_k = 0$ for some circuits and $CN_k - T_k > 0$ for the other circuits) -- the system performance just meets the given requirement.

(c) Some diagonal entries of matrix S are negative (i.e., $CN_k - T_k < 0$ for some circuits) -- the system performance is lower than the given requirement.

In the example, for $C = 15$, CP-Q is

	A	B	C	D	E	F	G
A	oo	oo	10	10	oo	oo	oo
B	oo	oo	10	10	oo	oo	oo
C	oo	oo	oo	oo	-5	oo	oo
D	oo	oo	oo	oo	oo	-4	oo
E	oo	-3	oo	oo	oo	oo	-3
F	oo	-3	oo	oo	oo	oo	-3
G	-2	oo	oo	oo	oo	oo	oo

After applying Floyd's algorithm to find the shortest distance between every pair of places we have:

	A	B	C	D	E	F	G
A	0	2	10	10	5	6	2
B	0	2	10	10	5	6	2
C	-10	-8	0	0	-5	-4	-8
D	- 9	-7	1	1	-4	-4	-7
E	- 5	-3	5	5	0	1	-3
F	- 5	-3	5	5	0	1	-3
G	- 2	0	8	8	3	4	0

Matrix S

Since the diagonal entries are non-negative, the performance requirement of $C = 15$ is satisfied. Moreover, since entries (A,A), (C,C), (E,E) and (G,G) are zero's, $C = 15$ is optimal (i.e., it is the minimum cycle time). In addition, when a decision-free system runs at its highest speed, CN_k equals to T_k for the bootleneck circuit. This implies that the places that are in the bottleneck circuit will have zero diagonal entries in matrix S. In the example, the bollteneck circuit is $At_1Ct_2Et_4Gt_5$. With this information, the system performance can be improved by either reducing the execution times of some transitions in the circuit (by using faster facilities) or by introducing more concurrency in the circuit (by introducing more tokens in the circuit). Which approach should be taken is application dependent and beyond the scope of this thesis.

The above procedure can be executed quite fast. The formulation of matrices P and Q takes $O(n^2)$ steps. The Floyd algorithm takes $O(n^3)$ steps. As a whole, the procedure can be executed in $O(n^3)$ steps. Therefore, the performance requirement of a decision-free system can be verified quite efficiently.

2.2.3 Safe Persistent Systems

A system is a safe persistent system if its Petri net model is a safe persistent Petri net. A Petri net is a safe persistent Petri net if and only if it is a safe Petri net and for all reachable markings, a transition is disabled only by firing the transition. It differs from a decision-free Petri net in that it may have more than one input (output) arcs to (from) a place. However, like a decision-free Petri net, it models a deterministic system. In a persistent Petri net, if a token enables a transition, it will be consumed by that transition only, i.e., a token will never enable two or more transitions simultaneously.

Figure 2.6a shows a persistent Petri net. It models the operations of a double buffer input port. Transitions t_1 and t_2 represent fetching the contents of buffer 1 and buffer 2 respectively. Transition t_3 represents storing the input into the memory. To compute the performance of the system, we first transform it into a decision-free system and then use the algorithm discussed in the previous subsection to compute the system performance.

A persistent Petri net can be transformed into a decision-free Petri net by tracing the execution of the system for one cycle. For example, Fig. 2.6b is the decision-free system corresponding to the persistent Petri net shown in Fig. 2.6a. Places A_1 and A_2 in Fig. 2.6b represent two different occurrences of place A in Fig. 2.6a in a cycle. Condition A_1 holds when transition t_1 is enabled. Similarly, place D is duplicated into D_1 and D_2. Condition D_1 holds when transition t_3 is enabled and transition t_2 will be enabled after firing t_3. Condition D_2 holds when transition t_3 is enabled and transition t_1 will be enabled after firing t_3.

Initially there is a token in places A and B and transition t_1 is enabled in Fig. 2.6a, and therefore there is a token in places A_1 and B in Fig. 2.6b. After firing t_1, a token is deposited into places C and D in Fig. 2.6a. This is represented by depositing a token in places D_1 and C in Fig. 2.6b. By following the execution of the system for a cycle, the corresponding decision-free system can be generated. The system performance can then be computed by the procedure discussed in Section 2.2.2.

2.2.4 General Systems

A system is a general system if its Petri net model is a general Petri net. A Petri net is a general Petri net if it is a consistent

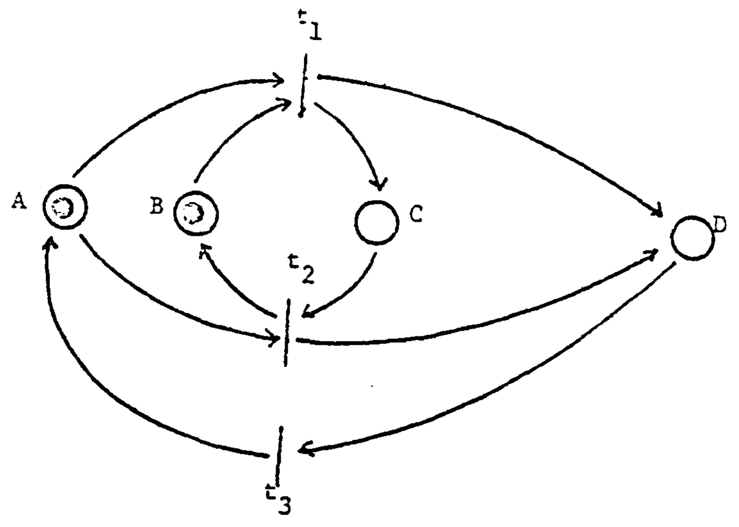


Figure 2.6a A persistent system

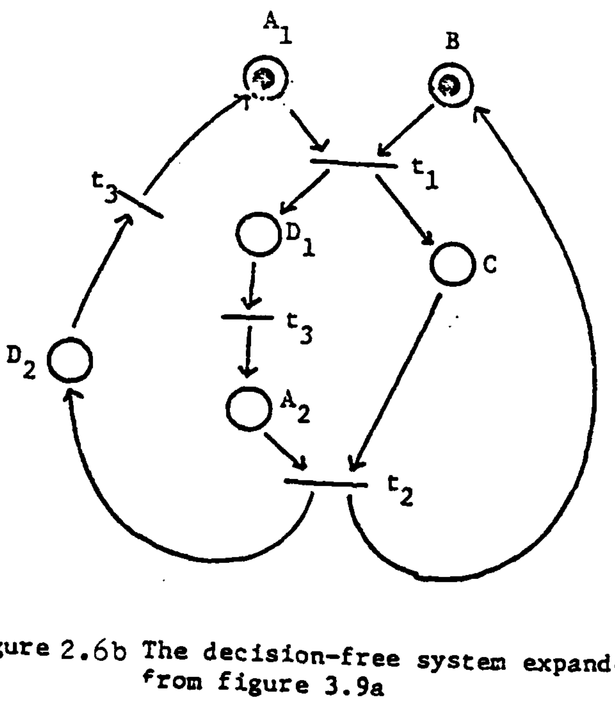


Figure 2.6b The decision-free system expanded from figure 3.9a

Petri net and there exists a reachable marking such that the firing of a transition disables some other transitions. Figure 2.7 shows a general Petri net. Figure 2.7 models the communication protocol from process P_1 to processes P_2 and P_3, such that the difference in the number of messages sent from P_1 to P_2 and P_3 is always less than three. It is not a decision-free Petri net becasue place A has more than one input and output arcs. It is not a persistent Petri net because, in the configuration shown, the execution of either transition t_2 or transition t_4 disables the other transition. This introduces the nondeterministic characteristic of the system.

General systems are very difficult to analyze. In the next theorem, we show that it is unlikely that a fast algorithm exists to verify the performance of a general system. A method of computing the upper and lower bounds of the performance of a conservative general system [Lie 76] is proposed. For a non-conservative general system, no good heuristics are known to the authors and further research is needed.

Theorem 2.4. Verifying the performance of a general Petri net is an NP-complete problem [Kar 72].

Proof:

(i) It is in NP because we can guess the otpimal schedule. The nondeterministisms of the general Petri net are resolved and the general Petri net can be transformed into a decision-free Petri net. The performance can then be verified by the procedure discussed in Section 2.2.2.

(ii) It is NP-complete because the set partitioning problem (an NP-complete problem) can be reduced to the above problem.

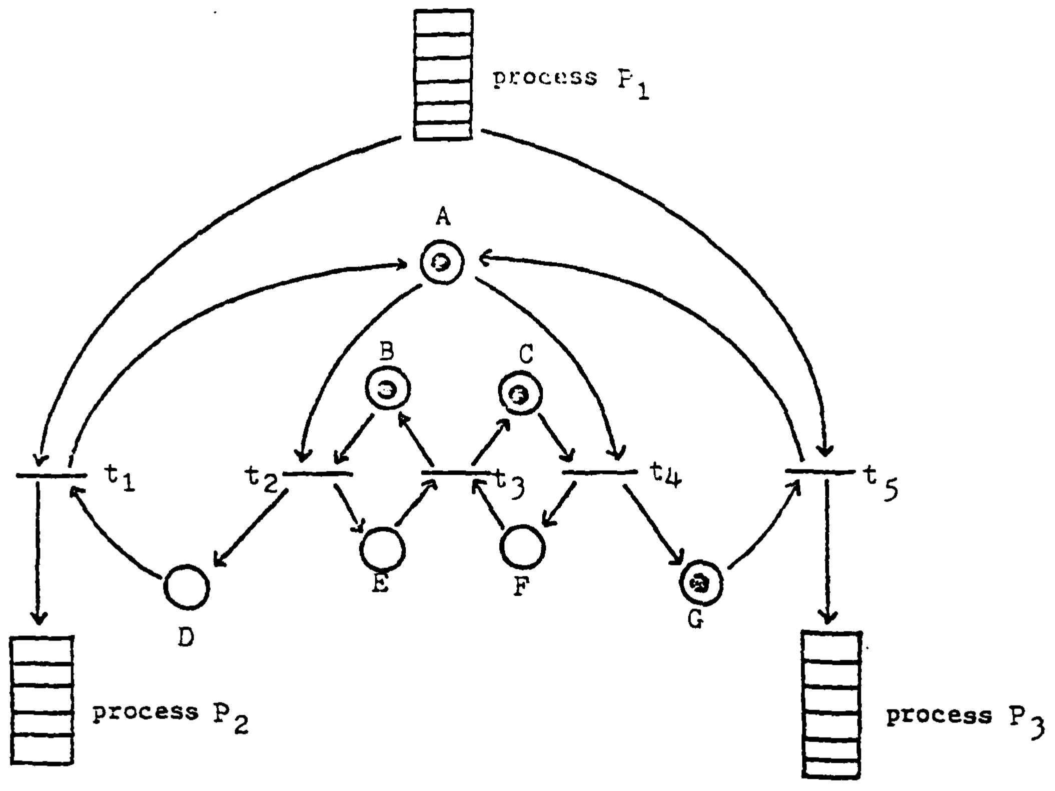


Figure 2.7 A general system (a communication protocol)

The Set Partition Problem [Kar 72]

Given a set of integers, $S = \{x_1, x_2, \ldots, x_n\}$, partition it into two subsets, S_1 and S_2 such that (condition B):

$$S = S_1 \cup S_2 \quad \text{and} \quad S_1 \cap S_2 = \emptyset \quad \text{and} \quad \sum_{i \in S_1} x_i = \sum_{i \in S_2} x_i = \frac{\sum_{i=1}^{n} x_i}{2}$$

Reduction

Given $S = \{x_1, x_2, \ldots, x_n\}$, generate the general Petri net shown in Fig. 2.8. It is easy to see that

the system has minimum cycle time, $C = \frac{\sum_{i=1}^{n} x_i}{2}$, if and only if the set S satisfies condition B.

A Method to Compute Upper and Lower Bounds of the Performance of A Conservative General System

(A) Upper Bound:

We choose a schedule which satisfies the execution frequency requirement and then the algorithm discussed in section 2.2.2 to find the cycle time of the system.

(B) Lower Bound:

(a) Find a non-zero integer assignment to the places such that the sum of integers assigned to the input places of a transition equals the sum of integers assigned to the output places. Such an integer assignment must exist because the system is conservative [Lie 76]. Intuitively, the integer assigned to a place represents the relative processing capability of a token at that place. The weighted execution time of a transition is equal to the produce of the transition execution time and the

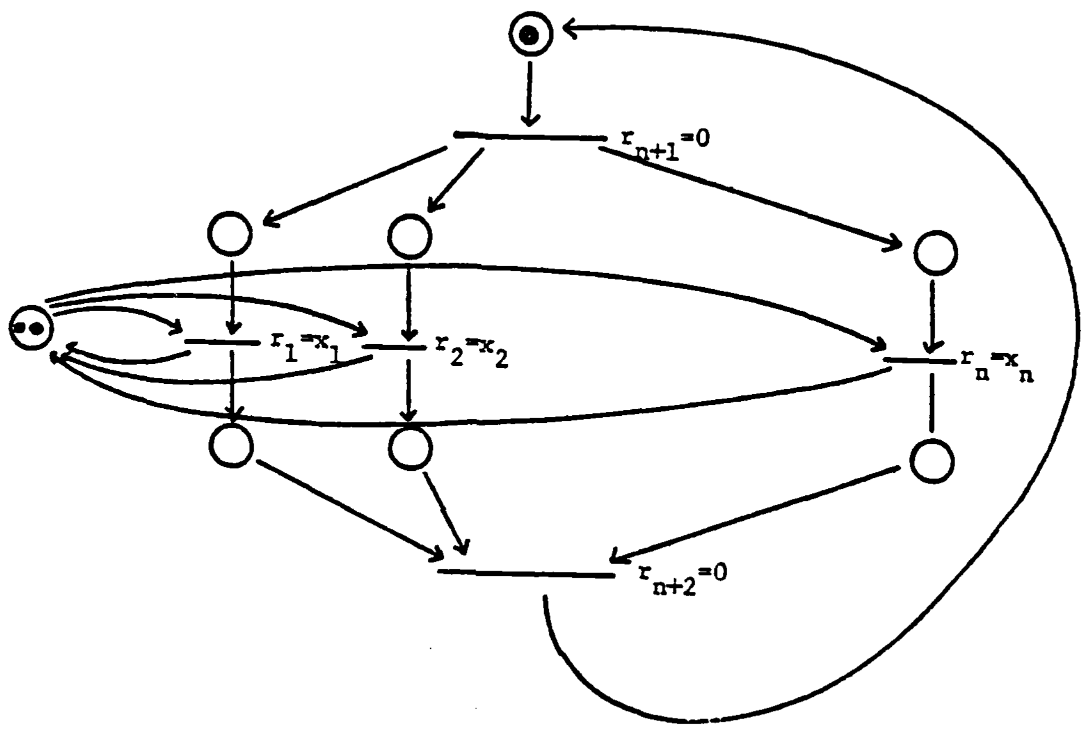
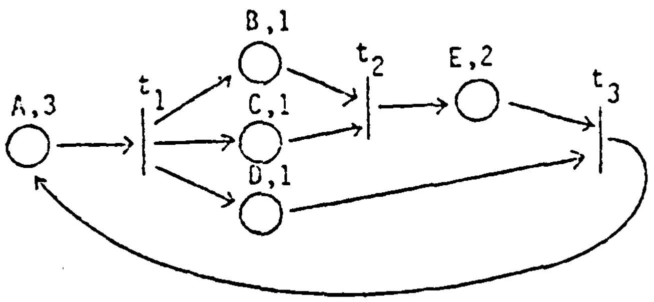


Figure 2.8 A reduction of the set partition problem to a general Petri net

sum of the integers assigned to its input places.



(b) Assume that all tokens in the net are busy all the time. Then

C x weighted processing capability $\geq$ sum of weighted execution time

$$C \times \quad s_i M_i \geq \sum s_i f_i r_i$$

$$C \geq \frac{\sum s_i f_i r_i}{\sum s_i M_i}$$

where s_i = sum of integers assigned to the input places of transition i

3. System Deadlock

3.1 An Approach to Deadlock Prevention

As pointed out in the introduction, the scope of our study on system deadlock is restricted to systems using: (i) binary semaphores; (ii) cricital regions; and (iii) monitors as their interprocess synchronization mechanisms. This enforces structural design and greatly reduces the computational complexities involved in the analyses. Based on the above synchronization constructs, a formal graph model (the request-possession graph) is developed to model deadlocks in these systems. The necessary and sufficient conditions for the occurrence of a deadlock are derived. Based on these conditions, techniques for uncovering potential deadlocks in a system are developed, and a systematic appraoch for the construction of deadlock-free systems using critical regions and/or monitors is proposed.

3.1.1 The Request-Possession Graph Model

An request-possession graph (an RP-graph) is a formal graph model developed to study deadlocks in systems which use binary semaphores, critical regions and/or monitors as their synchronization mechanisms. It is a directed bipartite graph with two types of nodes and two types of arcs (Fig. 3.1b): (1) resource reference nodes (which are called reference nodes in short in the rest of the chapter), and (2) resource nodes. The reference nodes are used to represent accesses of resources in a system and the resource nodes are used to represent resources. A dotted arc directed from a reference node to a resource node represents the request of the resource from the reference node. A solid arc directed from a resource to a reference node represents the assignment

Process X	Process Y
.	.
.	.
.	.
P(a)	P(b)
use printer	use tape drive
P(b)	P(a)
use tape drive	use printer
V(b)	V(a)
V(a)	V(b)
.	.
.	.
.	.

Figure 3.1a An example of system deadlock

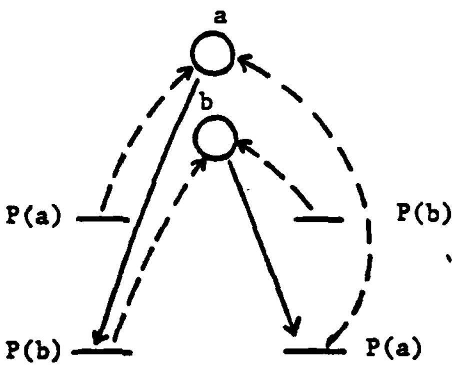


Figure 3.1b The request-possession graph
of figure 4.4a

of the resource to the reference node. From the above description, it seems that the RP-graph is very similar to Holt's general resource graph. However, they are quite different from each other. Holt's approach only models a snapshot of the resource state of a system while an RP-graph models the complete dynamic resource allocation characteristics of a system. The distinction will become clear later in our discussion.

The RP-graph of a program can be generated by scanning through the program once. The procedure for constructing the RP-graph of a concurrent system can be best illustrated by an example. Figure 3.1 shows a concurrent system together with its RP-graph. For each binary seamphore (P or V operation) in the system, there is a corresponding resource node (reference node) in the RP-graph. For each P operation in the system, a dotted arc is drawn from the corresponding reference node to the corresponding resource node. Solid arcs are then drawn from the resource nodes to a reference node for the resources that have been possessed by the process when it begins to execute the reference node. For example, solid arcs are drawn from resource nodes a and b to reference node V(b) of process X because both resources a and b are possessed by the process when it begins to execute instruction V(b). Following the above procedure, the RP-graph of a system can be constructed in linear time to the number of instructions in a program. As the releases of resources will never bring a system into a deadlock, the reference nodes corresponding to V operations are omitted in RP-graphs.

3.1.2 The Necessary and Sufficient Conditions for Deadlocks

The necessary condition for deadlocks developed in this section is applicable to systems using binary semaphores, critical regions and/or monitors as their synchronization mechanisms. The sufficient condition for deadlocks developed is only applicable to systems using critical regions and/or monitors as their synchronization mechanisms. This is due to the unstructureness of semaphores and are explained later in this section.

Definition. A system is safe if and only if it is deadlock-free. A system is unsafe if and only if it potentially can get into a deadlock state.

Theorem 3.1. If a system is unsafe, then there is a directed cycle in its RP-graph.

Proof: By definition, an unsafe system can potentially get into a deadlock state. In that state, there is a chain of processes such that each process holds one or more resources that are being requested by the next process in the chain [Cof 71]. Because of the way that the RP-graph is constructed, it can easily be seen that this chain of resource requests and resource possessions corresponds to a directed cycle in the RP-graph. Theorem 3.1 only gives the necessary condition for an unsafe system. The existence of a directed cycle in the RP-graph of a system does not imply that the system is unsafe.

The RP-graph can be generated automatically in linear time to the number of instructions in a system. The Floyd algorithm can be used to detect the existence of directed cycle in the RP-graph, which has execution time $O(n^3)$ steps where n is the number of nodes in the generated RP-graph. As a result, the proposed algorithm can be executed in polynomial time to the number of instructions in a system.

Before we discuss the sufficient condition for a safe system, some extensions have to be made on the RP-graph. The resultant model is called the augmented request-possession-graph (the ARP-graph). It is very similar to the RP-graph except that each reference node, r, is given a set of names, S_r, such that $s \in S_r$ if and only if:

(1) s is the name of the process when it begins to execute node r

or (2) s is the name of a resource that has been possessed by the process when it begins to execute node r (i.e., there exists a solid directed arc from resource s to node r in the RP-graph).

Theorem 3.2. A system is safe if and only if its ARP-graph does not contain a directed cycle with distinct names on its reference nodes (i.e., $S_u \cap S_v = \emptyset$ for all pairs of nodes, u and v, in the cycle).

Proof: Let P be the proposition that there is a directed cycle with distinct names in an ARP-graph. For a given directed cycle in an ARP-graph, define the state of a process as the state at which the process begins to execute the instruction which corresponds to the reference node of the process in the cycle. Define the system state as the state composition of all the processes in the system.

(A) The Sufficient Condition

$\neg$ P $\Rightarrow$ System is safe

i.e. System is unsafe $\Rightarrow$ P

System is unsafe $\Rightarrow$ there exists a deadlock state, q

$\Rightarrow$ there exists a directed cycle in the RP-graph (Theorem 3.1)

$\Rightarrow$ P (because no resource can be possessed by two processes in state q)

(B) The Necessary Condition

System is safe $\Longrightarrow$ -P

We first assume that a system is safe and P, then show that there is a contradiction. Let state q be the system state that corresponds to the directed circuit in the ARP-graph. If the system is safe and P, there are two cases:

(i) State q is reachable from the initial state:

This means that the system can potentiall get into a state in which there exists a chain of processes such that each process holds one or more resources that are being requested by the next process in the chain. That is, the system can potentially get into a deadlock state. This produces a contradiction.

(ii) State q is not reachable from the initial state:

Without loss in generality, consider the case shown in Fig. 3.2. Assume that in state q, process X is in region c, process Y is in region f and process Z is in region i. It is easy to see that in order to reach state q, region e in process Z has to be executed before region e in process Y. This type of precedent relationships are shown by the arrows in the figure. Assume that there are implicit arrows pointing from one instruction to the next instruction in a process. There are two sub-cases to be considered:

(a) There is no directed cycle in the graph:

This implies that there exists an execution sequence that can bring the system into state q. In particular, the sequence can be constructed by executing ready instructions (i.e., instructions whose parent instructions have been executed) of a process as far

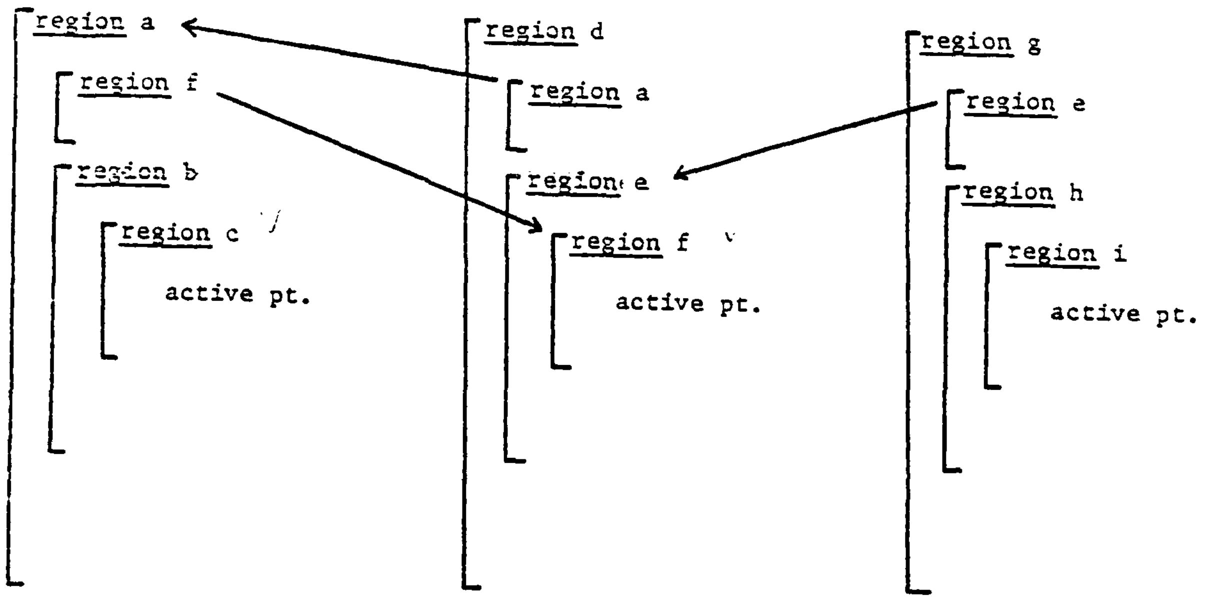


Figure 3.2 A deadlock-free system

as possible until the process comes to an instruction which has an unexecuted parent instruction. The well-structured property of critical regions and monitors guarantees that a process releases all the resources (except those it possesses in state q) that it has acquired during the execution before the process is blocked (condition A). This guarantees that the execution will not create any new blockades due to resource sharing to the following executions. The acyclic graph property guarantees that there always exists some process ready to be executed until the system comes to state q. As a result, state q can be reached and this produces a contradiction.

(b) There is a directed cycle in the graph:

This means that condition P holds at an earlier stage in the execution. By repeating the above procedure, either condition (i) or condition (a) will hold finally. This produces a contradiction.

Theorem 3.2 is true for a system which uses critical regions and monitors as its synchronization mechanisms, however, it does not hold for a system that uses semaphores as its synchronization mechanism. From this point onwards, when we talk about systems, we mean systems which use critical regions and/or monitors as their synchronization mechanisms.

One application of theroem 3.2 is to prove the safety of a system. Before we can use the Theorem, we have to develop an effective procedure to determine whether there exists a directed cycle with distinct labels on its nodes in a labelled directed graph. However, it is shown in the following theorem that the above problem is NP-complete (i.e. it is unlikely to have a fast algorithm to solve the problem).

Theorem 3.3. It is NP-complete to find a directed cycle with distinct labels on its nodes in a labelled directed graph.

Proof:

(A) The problem is NP because we can guess a directed cycle that satisfies the requirement and verify our guess by tracing the cycle in the graph in polynomial time to the number of nodes in the graph.

(B) The problem is NP-complete because the 2-3 satisfiability problem (an NP-complete problem) can be reduced to the above problem.

The 2-3 satisfiability problem [Aho 76]:

To find whether there exists a satisfying assignment (an assignment that gives the expression a true value) to a boolean expression which has the following properties:

(a) every clause contains three literals

(b) every variable x appears exactly twice as x and twice as $\bar{x}$

Reduction

Let $S = C_1 C_2 \ldots C_p$ be an instance of the 2-3 satisfiability problem. Assume that clause C_i contains variables y^1, y^2 and y^3. Construct a labelled directed graph, G, as shown in Figure 3.3.

If $y^i = x$ then label: $y^i_a = x_1$, $y^i_b = x_2$, $y^i_c = x_3$ and $y^i_d = x_4$

If $y^i = \bar{x}$ then label: $y^i_a = x_1$, $y^i_b = x_3$, $y^i_c = x_2$ and $y^i_d = x_4$

Claim:

Expression S is satisfiable if and only if there is a directed cycle with distinct labels on its node in graph G.

Theorem 3.3 states that it is NP-complete to determine the safety of a system. However, if the number of critical regions in a system is fixed, the computation complexity involved in determining the safety of the system is in polynomial to the size of the system (where the size of a system is defined as the number of reference nodes, n, in its

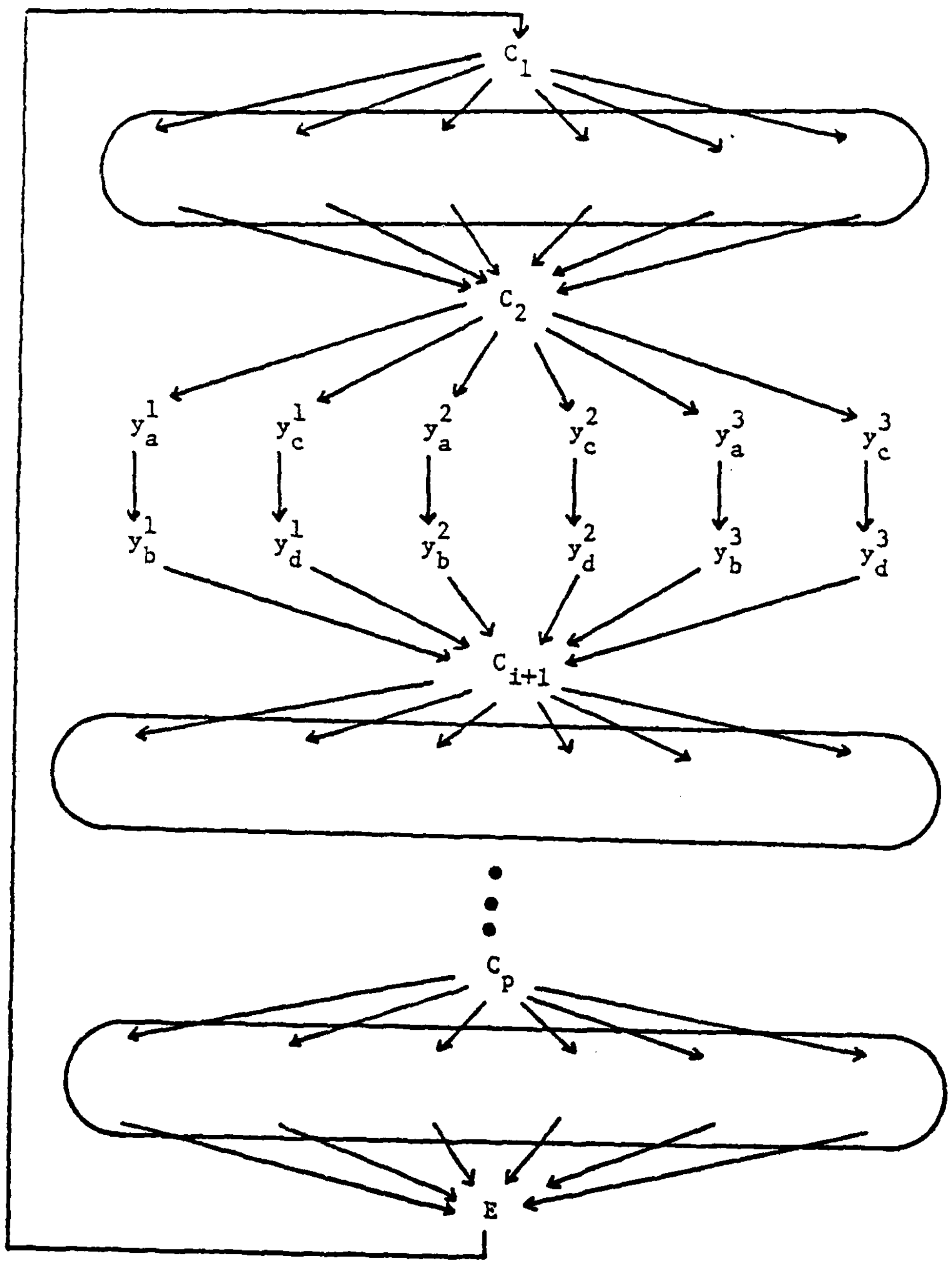


Figure 3.3

ARP-graph). It is because there are at most k nodes in a cycle which has distinct names on its nodes. As a result, there are at most 2^k such cycles. By trying to construct such a cycle from each reference nodes in the ARP-graph, we can verify the safety of a system in $O(2^k n^3)$ steps. The details of the procedure will not be discussed in this report

3.1.3 An Approach to the Design of Deadlock-Free System

Theorem 3.4

If all critical regions and /or monitors are linearly ordered, and all processes enter a critical region or a monitor at a higher level before those at a lower level, deadlock cannot occur.

Proof:

The theorem is a special case of the theorem on resource allocation given in [Hav 68]. By the imposition of a linear ordering on the critical regions and/or monitors, a circular wait cannot occur and therefore the system is deadlock-free.

The above strategy imposes severe constraints on the nesting of critical regions and/or monitors. Two of its drawbacks are: (1) reducing the concurrency in a system; and (2) reducing the transparency of a system. One approach to remedy some of the drawbacks is to group critical regions and/or monitors into sets allowing unordered nesting within each set. A linear ordering is then imposed among sets. A process must not enter a critical region or a monitor in a set at a higher level after it has entered one in a set at a lower level. The linear ordering among sets guarantees that deadlock cannot occur due to improper nesting of critical regions or monitors in different sets. The deadlock-free condition within each set is verified by the deadlock detection procedure discussed in section 3.1.1. This approach provides: (1) good programming style; (2) higher degree of concurrency; (3) no run time overhead; and (4) automatic deadlock detection during compilation.

3.2 Deadlock Detection in Distributed Data Bases

In a distributed data base, deadlocks can be detected quite easily by using a centralized control strategy. Whenever a process locks or releases a data file, it gets the permission from a central control node. This control node maintains a demand graph for the whole system and checks for deadlocks by searching for a directed cycle in the graph. However, the approach is inefficient. All data accesses have to get the permissions from the central control node although they may not cause any deadlocks. This slows down the system, wastes the system communication bandwidth and unnecessarily congests the communication subsystem. Above all, if the control node goes down, it is very difficult to recover the system from the failure.

Another approach for deadlock detection is to store the resource status locally at each site. Periodically, a node is chosen to be the control node. Resource status are then sent from each site to the control node for analyses. This remedies most of the drawbacks of the centralized approach. However, due to the inherent communication delay, the chosen control node may get an inconsistent view of the system, and it may make a wrong conclusion.

We have developed three approaches to construct consistent demand graph. In the approaches, it is assumed that each transaction is given a unique name. (A transaction is defined as a sequence of actions which can be a request, lock or unlock operations.) This can be achieved by naming a transaction with the name of its site of origin together with the initiation time of the transaction [Ber 78].

3.2.1 A Two Phase Deadlock Detection Protocol

In this protocol, each site maintains a status table for all resources that are owned by the site. For each resource, the table keeps track of the transaction that has locked the resource (if one exists) and the transactions which are waiting for the resource (if they exist). Periodically, a node is chosen as the control. The chosen control node performs the following operations:

(1) Broadcasts a message to all nodes in the system requesting them to send their status tables and waits until all tables have been received.

(2) Constructs a demand graph for the system:

- (a) If there is no directed cycle, the system is not in a deadlock and the node releases its control.
- (b) If there is a directed cycle, the node continues its execution.

3) Broadcasts a second message to all nodes in the system requesting them to send their status tables and waits until all tables have been received.

(4) Constructs a demand graph for the system using only transactions that are reported in both the first and the second reports:

- (a) If there is no direct cycle, the system is not in a deadlock and the node releases its control.
- (b) If there is a directed cycle, the system is in a deadlock. The node reports the deadlock situation to a deadlock resolver.

The above procedure uses a two phase commit protocol. By only using transactions that are reported in both the first and the second status reports in constructing the demand graph, a consistent system state is obtained. The main advantage of this protocol is its simplicity. The drawback is the requirement of two status reports from each site before a deadlock can be determined. In general, the protocol is good for systems in which deadlocks occur only infrequently.

3.2.2 A One Phase Deadlock Detection Protocol

In this protocol, a deadlock is detected in one communication phase. Each site maintains a resource status table for all local resources and a process status table for all local processes. The resource status table keeps track of the transactions that have locked a local resource and the transactions which are waiting for a local resource. The process status table keeps track of the transactions that are being owned by processes local to the site. The system operates according to the following rules:

(A) A process at site S requests a resource -- a transaction (S,t) is created, where S is the site name and t is the time at which the transaction is initiated. An entry (S,t,w) is put into the process status table of the site indicating that transaction (S,t) is waiting for a resource. A message is sent to acquire the resource.

(B) Site T receives a message that transaction (S,t) requests a resource local to T:

  (i) If the resource is free, the resource is assigned to the transaction and a lock is set on the resource. An entry (S,t,a) is created in the resource status table of the site and a message is sent to notify the requesting process of the assignment.

  (ii) If the resource is being locked, an entry (S,t,w) is created in the resource status table of the site and a message is sent to acknowledge the receiver of the request.

(C) Site S receives a resource assignment message for transaction (S,t) -- the entry (S,t,w) in the process status table is changed to (S,t,a).

(D) Site S receives a request acknowledgement message for transaction (S,t) -- do nothing.

(E) A process at site S releases a resource corresponding to transaction (S,t) -- the entry (S,t,a) is removed from the process status table and a message is sent to notify the release.

(F) Site T receive a resource message corresponding to transaction (S,t) -- the resource is unlocked and the entry (S,t,a) is removed from the resource status table.

Periodically, a node is chosen as the control. The chosen control node performs the following operations:

(1) Broadcasts a message to all nodes in the system requesting them to send their status tables and waits until all tables have been received.

(2) Constructs a demand graph for the system using only transactions for which the resource status table agrees with the process status table (i.e. identical entries exist in both the resource status table and the process status table).

    (a) If there is no directed cycle, system is not in a deadlock and the node releases its control.

    (b) If there is a directed cycle, system is in a deadlock. The node reports the deadlock situation to the deadlock resolver.

In order to show that the above protocol is correct, we have to prove that the existence of a directed cycle in the constructed demand graph implies the occurrence of a deadlock state.

Theorem 3.5

A system is in a deadlock if and only if there is a directed cycle in the demand graph constructed by the above procedure.

Proof:

(A) The necessary condition

If a system is in a deadlock, there is a chain of processes such that each process locks a file which is being requested by the next process in the chain. This condition will remain valid until the deadlock is resolved. This implies that there is a directed cycle in the constructed demand graph although the information are collected at different times.

(B) The sufficient condition

Assume that there is a directed cycle in the constructed demand graph and assume that there is a master who keeps track of the absolute times of the occurrences of the activities in the system. (The absolute times are used in the proof only and are not needed in the protocol). With no loss in generality, assume that the resources and the processes involved in the cycleare at different sites in the system. There are two cases to be considered:

(i) The latest report is a process status report (Figure 3.4)

Assume that t_g is the latest time. From the resource status report of R_1 and the process status report of process P_1, we know that resource R_1 is possessed by process P_1 from time t_i to t_g. From the resource status report of R_1, we know that process P_2 is waiting for resource R_1 at time t_i. Since resource R_1 is possessed by process P_1 from time t_i to t_g, process P_2 is still waiting for resource R_1 at time

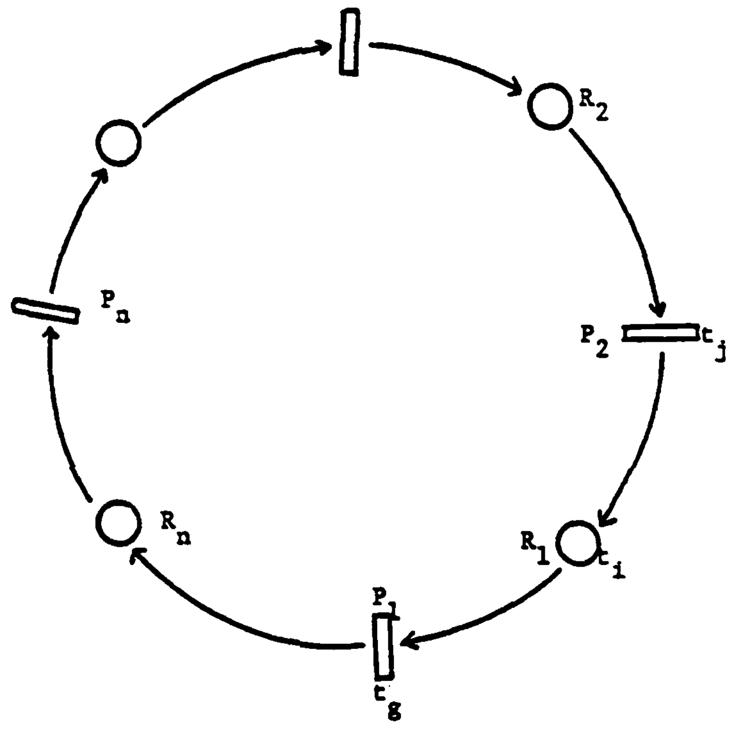


Figure 3.4

t_g. By continuing this argument, resource R_n is possessed by process P_n at time t_g. As a result, at time t_g, there is a chain of processes such that each process holds one resource that is being requested by the next process in the chain. This implies that the system is in a deadlock.

(ii) The latest report is a resource status report

The argument is similar to that given in (i).

The one phase deadlock detection protocol does not need a second communication phase to confirm the occurrence of a deadlock. However, more information has to be stored at each site and more data have to be transferred in a communication phase. The protocol is good for systems in which deadlocks occur only infrequently and the communication cost depends on the number of messages being sent rather than the length of the messages.

3.2.3 A Hierarchical Deadlock Detection Protocol

In very large distributed data bases, it may be very costly to transfer all status tables to one site. In particular, if the access pattern is very localized, it will be of great advantage if deadlocks are detected locally. In these systems, one approach is to group sites which are close to each other into a cluster. Periodically, a node in a cluster is chosen to be the control. This control node executes the one phase deadlock detection protocol and constructs a demand graph for the cluster. The result obtained by the control node together with the intercluster accesses (which should be relatively few) are then sent to a central control node (which is also chosen dynamically). Based on these information, the central control node constructs the demand graph of the whole system. In this way, deadlocks within a cluster are detected by the control node of the cluster and deadlocks among clusters are detected by the central control node.

Defintion

A transaction is a local (intercluster) transaction if and only if the requesting process and the requested resource are in the same (different) cluster(s).

A Hierarchical Deadlock Detection Protocol

(A) Periodically, a central control node is chosen. This node performs the following operations:

  (1) Chooses dynamically a control node for each cluster.

  (2) Broadcasts a message to all control nodes requesting them to send their status information and wait-for relations of the intercluster transactions.

  (3) Constructs a demand graph of the system using both the intercluster transactions for which the resource status report agrees with the process status report and the wait-for relations (which are defined later) sent from the control nodes. If there is a directed cycle in the demand graph, the system is in a deadlock, otherwise, the system is not in a deadlock.

(B) Whenever a node recieves a status report request from the central control node, it performs the following operations:

  (1) Broadcasts a message to all nodes in the cluster requesting them to send their status tables and waits until all tables have been received.

  (2) Constructs a demand graph for the cluster using only local transactions for which the resource status table agrees with the process status table.

(3) Computes the transitive closure of the demand graph. If there is a directed cycle in the demand graph, the system is in a deadlock.

(4) Derives the wait-for relations from the transitive closure of the demand graph. A process/resource is waiting for a process/ resource if and only if:

   (a) the processes and/or the resources are in some intercluster transactions.

   (b) The process/resource is waiting directly or indirectly for the process/resource (i.e. there is a directed arc pointing from the process/resource to the process/resource in the transitive closure of the demand graph).

(5) Send the intercluster transaction status information and the wait-for relations to the central control node.

The above concept can be extended into many levels. In this way, a hierarchy of control nodes can be constructed. Due to the local access pattern of a system, the amount of information that have to be sent from a child control node to its parent can be greatly reduced.

4 Classification of Failures

4.1 Classification Methods

In order to develop generalized techniques and desired architectural features for failure detection, isolation, and recovery, it is essential to classify the failures further into several categories that reflect the generalized failure detection and recovery techniques. Such classification can be done based on several of the following criteria. These are 1) failure detection and correction, 2) physics of failures, and 3) consequences of failures in relation to system design and operation. The first method of classification divides the failure modes into different categories, each having the same or similar failure detection and correction mechanisms. Such classification is dependent on the existing system architectures and their implementations and does not really help the designer understand the behavior of the system to improve its fault tolerance.

In the second method of classification, which is based on the physics of a failure, the physical behavior of a component under investigation is considered in order to determine the nature and causes of a failure. This classification is quite satisfactory at a component level but fails to consider the logical failures and the failures at system and subsystem levels because of the innumerable physical parameters involved.

4.2 Proposed Classification

In order to alleviate the problems associated with these two methods of classification, we adopt a classification technique based on the consequences, or the observed effects and pragmatic considerations. In this classification we consider the consequences of a failure rather than its causes, since it is the effect of a failure that is eventually detected.

In this way, all the failures having the same observable effect could be considered uniformly. Such classification facilitates the use of the same failure behavior models for all failures producing the same net effect irrespective of their origin. Also, this approach permits the comparison of several system configurations based on the observed consequences of failures.

The first dimension of our classification is based on the level of the system being considered:

* system level (e.g. distributed system)
* subsystem level (e.g. node or computer level, job)
* module level (e.g. I/O, CPU, memory, task)
* functional submodule level (e.g. multipliers, subroutine)
* micro module level (gates, instructions)
* base level

A system is considered to have a set of subsystems each of which consists of a set of modules, which in turn consists of a set of sub-modules and so on. This level classification is the same for both the hardware system as well as the software system. Such a view allows a uniform classification of a system's failures independent of its final implementation details. At any given level of a distributed system, the failures are further classified into three different major classes (i) failures associated with the processing subsystem (A_0), (ii) failures associated with the communication subsystem (B_0) and (iii) the failures associated with application functions and the environment (C_0). The failures associated with each of the above categories can be further classified based on the physical, logical characteristics related to both design and operation failures of a distributed system. Fig. 4.1 provides a detailed breakdown of one class of processing logic failures viz. memory failure.

Our classification methodology is a generalized failure classification for distributed systems. Such classification allows implementation independent failure behavior of a system. This classification also reduces the number of failure cases to be analysed in the study of reconfigurability of a system.

4.3 Reconfiguration

The design of dynamically reconfigurable architectures is basically driven by two factors. One is the need to match the demands of the load to the capacity of the system by adapting the configuration to the load on hand. Second is the issue of enhancing the reliability and availability of a system by changing its configuration so that the effect of the failure is bypassed. This ability of a system known as reconfigurability can be degined as the systems ability to change its physical and/or functional organization in response to changing processing requirements of an application or to the failures of a system.

4.4 Methods of Reconfiguration

Basically, the reconfiguration strategies suitable for either type of reconfiguration could be classified as i) static, ii) dynamic, and iii) adaptive reconfigurations. In static reconfiguration, the change in the system configuration may not be achieved on line. Generally, such reconfiguration is done by suspending the ongoing system operations and restarting the system after the desired configuration is achieved. In this static case, the current as well as future configurations and the

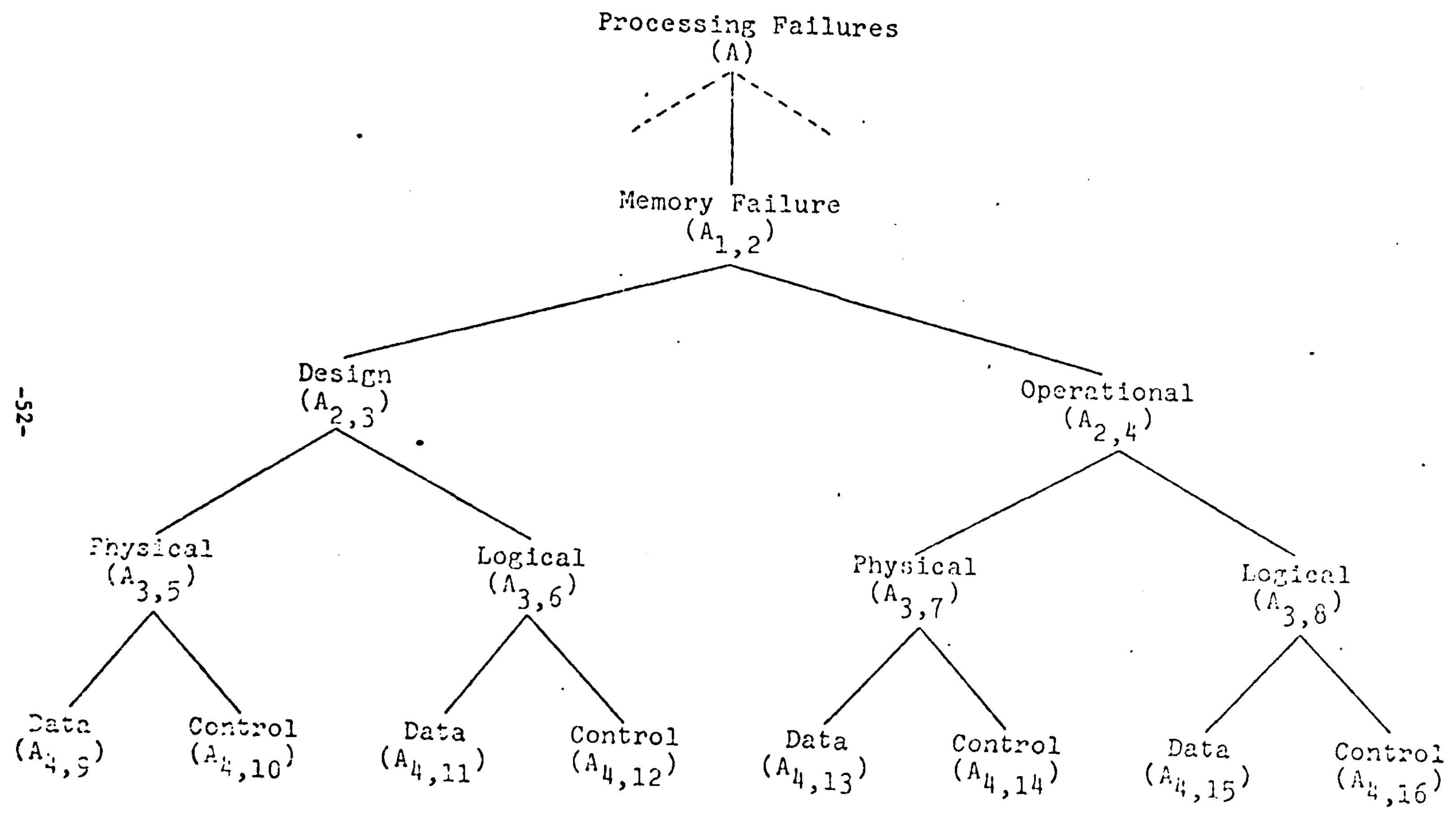


Fig. 4.1 - Processing Failures

transition involved are known in advance. One great drawback of static reconfiguration schemes is that they often involve significant interruption of service.

In Dynamic Reconfiguration, the changes in system configuration are initiated without much interruption to the service. In other words, such reconfiguration is done without bringing the system to a halt and is usually transparent to the user. Since this is done on a dynamic basis the status of the system after reconfiguration may not be known in advance. However, the strategies for a given situation, whether it is recovery from a failure or changes in the processing requirements, are known in advance. Such dynamic reconfiguration has been studied extensively by several authors [KART77, MORT74, REDD78, SCHE71].

Adaptive Reconfiguration is a generalized case of dynamic reconfiguration. In this case the process of reconfiguration tries to adopt a particular strategy based on the current status of the system and implements a selected strategy without any interruption to the service. In other words, the strategy selection and its implementation are not known in advance and the mechanism adapts to the situation so that these mechanisms are transparent to system operation. This type of reconfiguration offers the advantage of tailoring the machine to a given environment, thus allowing optimum use of the resources. Our report deals with this method of reconfiguration which paves the way to the ultimate goal of self repairable, self-adaptable modular distributed computer systems.

5 Design of Reconfigurable Systems

5.1 Proposed Design Methodology

Our methodology (figure 5.1), divides the process of designing any large scale distributed processing system into four major phases. They

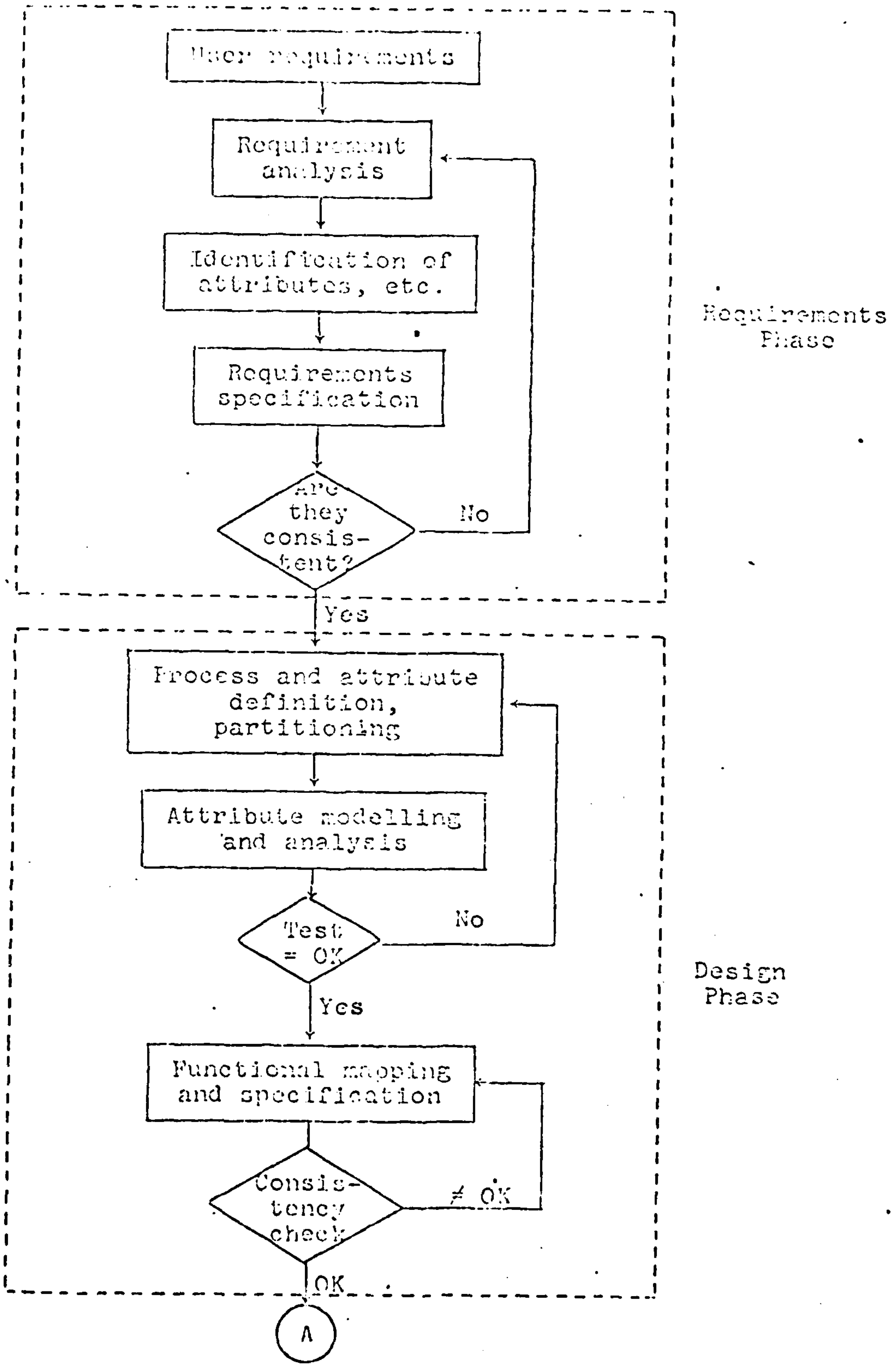

User requirements
Requirement analysis
Identification of attributes, etc.
Requirements specification
Are they consistent?
No
Yes
Requirements Phase
Process and attribute definition, partitioning
Attribute modelling and analysis
Test = OK
No
Yes
Functional mapping and specification
Consistency check
≠ OK
OK
A
Design Phase

Fig. 5.1 - Proposed Design Methodology

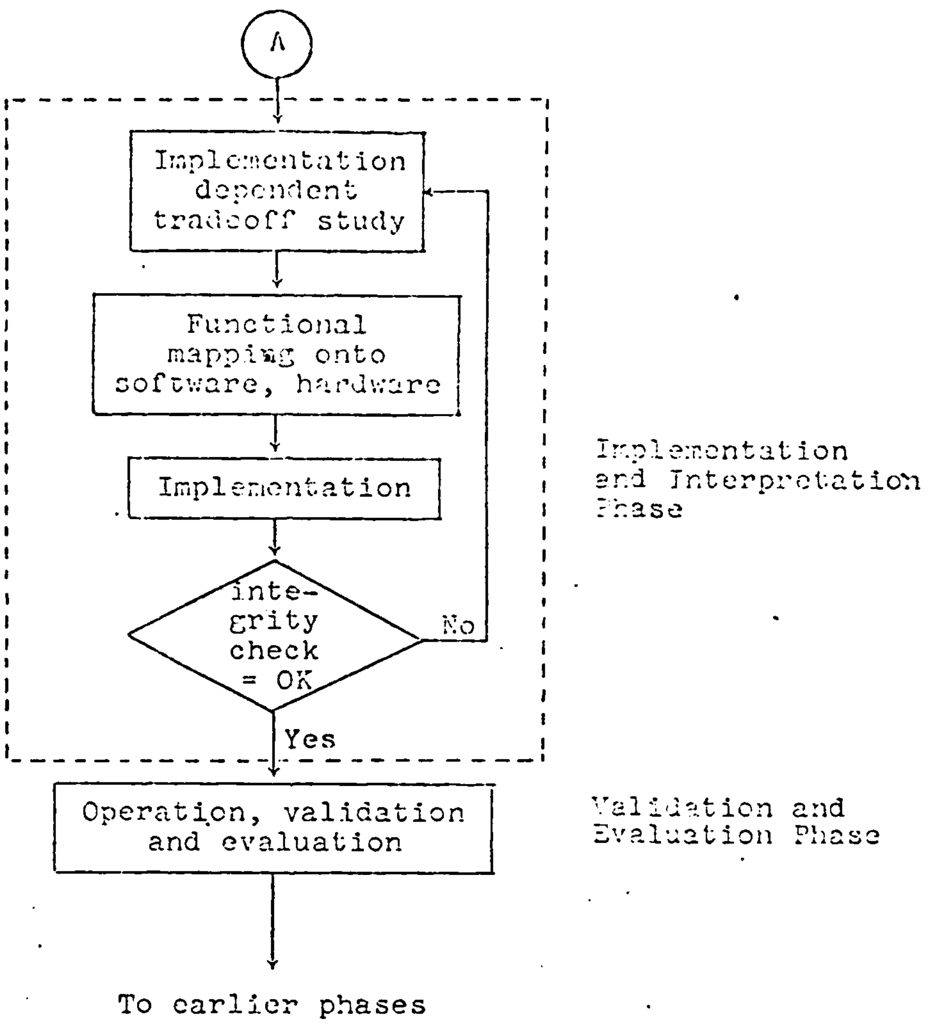

A
Implementation dependent tradeoff study
Functional mapping onto software, hardware
Implementation
integrity check = OK
No
Yes
Operation, validation and evaluation
To earlier phases
Implementation and Interpretation Phase
Validation and Evaluation Phase

Fig. 5.1 (continued)

are i) requirements and specification phase, ii) design process phase, iii) implementation and integration phase and iv) validation and evaluation phase.

Requirements and Specification Phase

The "requirements and specification phase" of the methodology starts with informally specified user requirements and elaborates them in terms of the design specifications. The first step in this phase is requirement analysis, that deals with the decomposition and translation of the users' needs and objectives into system engineering terminology. For example, the overall requirements of a system can be decomposed into data processing requirements, resource allocation and communication requirements, reliability, availability requirements, and so on. These requirements can further be decomposed into low level functional requirements. For example, reliability, and availability requirements can be further decomposed into user requirements, recovery requirements, resource requirements and so on. Some studies have been performed on formalizing and automating the requirement analysis and design for large scale software systems [PARN72, BOEH74].

The second step in the requirements phase is the specification of the requirements. This step is concerned with the logical aspects of a DCS and generates a set of specifications based on the analyzed requirements and attributes. For the successful execution of this phase, a formal manner of specifying these requirements in terms of a specification language is needed. The process of the specification also requires the identification of various processes needed to support the required functions of the system, the system parameters, and the attributes, in order to test the functional consistency of a system design. This enables the early detection of certain system design errors.

Design Phase

The second major phase of our methodology is the design phase. The design phase starts with the defined processes which are the outputs of the requirements and the specification phase. The major steps involved in the design phase are partitioning of the attributes and the functions, development of the appropriate models and their analyses, specification of abstract functional modules, and design verifications.

In the attribute and functional partitioning step, the designer will decompose the system into a set of interacting functions, and identify the associated attributes consistent with the overall design requirements, and their elaborations. The motivation for such partitioning is twofold. First of all, functional partitioning increases the modularity and testability and decreases the interference between processes. Secondly, the attribute decomposition allows the isolation of important attributes from each other.

The second step in the design phase of the methodlogy is the development of appropriate models and analysis techniques for the attributes. This report is mainly concentrated on this step of the design methodology. Specifically, it deals with the modelling and analysis of dynamic fault tolerance (reconfiguration) and other related attributes, as discussed in subsequent sections.

The third step in the design phase is the functional mapping and the specification of abstract functional modules. The functional mapping and specification refers to the definition of a partitioned process in terms of input/output relations which reflect the functional requirements and their integration. These specified functions are then integrated based on their coupling and other criteria related to the attributes, in an effort to check their consistency and to optimize them. This logical design is the input to the implementation phase.

Implementation and Integration Phase

The third major phase of our methodology is the implementation and integration phase. In this phase the functional modules are considered together with the additional implementation dependent design constraints which were not considered in earlier phases. This phase also includes the trade-off studies to determine what functions can be implemented in hardware and what others can be implemented in software and/or in firmware.

Another phase of our methodology the validation and evaluation phase, considers the operation of the system and evaluates various attributes to see whether they conform to the users' needs. If they do not, then appropriate modifications are initiated at the appropriate points.

The design methodology discussed above will be illustrated in Section 7 of this report.

5.2 Attribute Analysis using UGM

In order to design a reconfigurable distributed system there are several attributes that need to be modelled and analysed during the design phase e.g. reliability and performance. Proper modelling and analysis of each of these attributes is the key to the development of a true design methodology. We now present the Unified Graph model, which is a generalization of the UCLA model, as a suitable modelling technique for analysing these attributes.

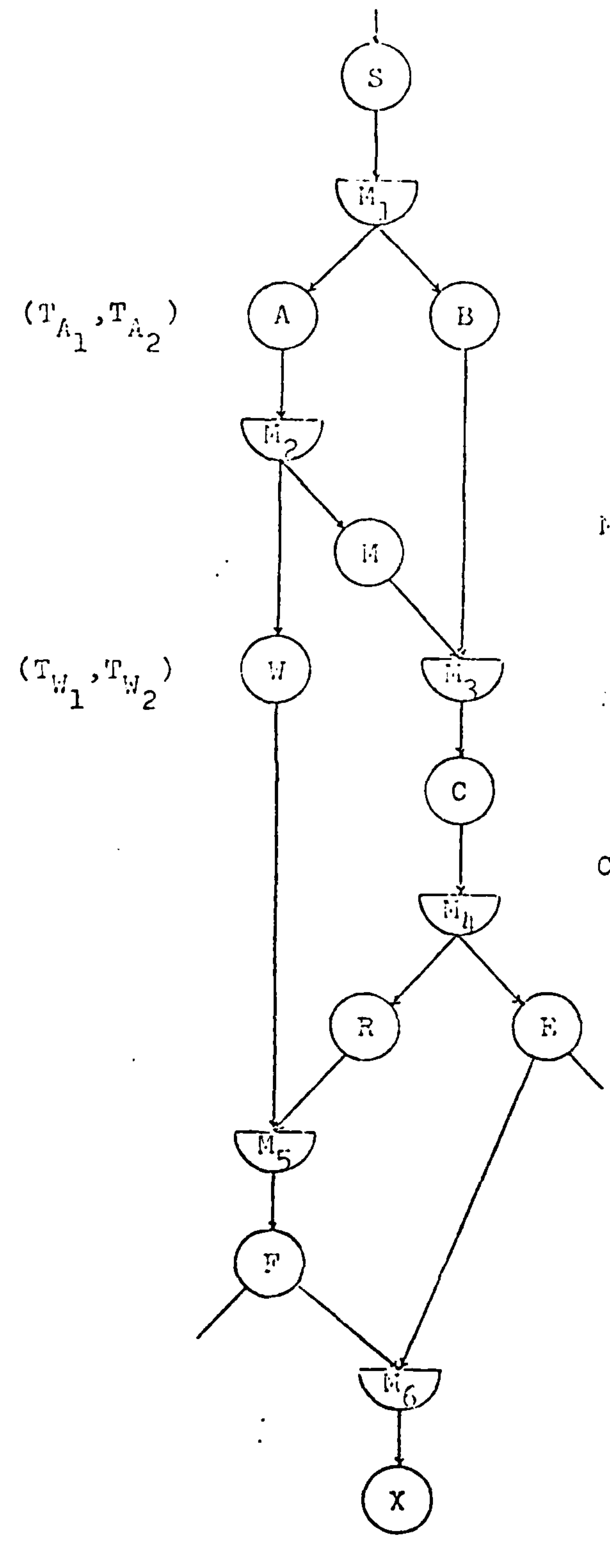
Definition 5.2.1:

A Unified Graph Model (UGM) is a Four tuple <G,M,C,T> where

1. 'G' is a directed graph
2. 'M' ($\overleftarrow{M},\overrightarrow{M}$) is a mapping function from a set of vertices $v_i \in V$ into a set $v_o \in V$ defined by a Boolean function (AND, OR, sum of products). $\overleftarrow{M}(v)$ specifies the logical conditions under which vertex 'v' can initiate execution. $\overrightarrow{M}(v)$ specifies the logical consequence of the completion of execution of vertex v. In the context of UGM, the execution of a vertex implies a transition from one vertex to another releasing a token.
3. 'C' is a set of coordination constraints on set V.
4. $T = \{T_{I1}, T_{I2}\}$ represents the max, min execution time of each v that belongs to V.

Figure 5.2 is an example of a UGM representation of the exchange of communication between sender and a receiver. M_1, M_2, M_3, M_4 are the mapping functions defining various Boolean relations among the nodes. In this particular example the process of exchanging communication is coordinated through the coordination constriants defined by the nodes 'S' and 'X'. The constraint 'S' defines that the nodes A and B should hold the tokens (i.e. sender is in SEND mode, and receiver is in RECEIVE mode) simultaneously for the two members to communicate. Similarly, for the system of figure 5.2 to reach the state 'X', the vertices F and E should have the tokens simultaneously and is defined by the coordination constraint X --> F.E. However, complex cases of coordination arise in distributed system and only the coordination constraints defined on the set 'V' define the interrelationships and interactions between various autonomous or semiautonomous subsystems.

Informally, the set of nodes 'V' represents the actions or computations to be performed or the status of the individual subsystems or



S - start

A - sender ready

B - receiver ready

W - wait for acknowledgement

M - message sent

C - message received

R - acknowledgment sent

E - message arrival signalled

F - message delivery

Mapping functions:

M_1	$S \rightarrow A+B$
M_2	$A \rightarrow W \cdot M$
M_3	$B \cdot M \rightarrow C$
M_4	$C \rightarrow R \cdot E$
M_5	$R \cdot W \rightarrow F$
M_6	$F \cdot E \rightarrow X$

Coordination constraints:

S - A·B

X - F·E

Fig. 5.2 - UGM Representation of Exchange of Messages

components. The set 'V' can in general be defined at any level of abstraction.

The mapping function M specifies the logical condition under which a vertex 'v' can initiate execution. The mapping function relates the activities of the input node set and the associated arcs, to the activities of the output node set and the associated arcs on a node 'v'.

The coordination constraints 'C' define the degree of coupling between various subsystems. This set 'C' limits the possible states to states of importance only. It designates what events may coexist with the same node, and what set of inputs may be used in a given situation. There may be several events, but only a set of them is applicable during a particular observation event (definition 5.2.2).

Definition 5.2.2

"Event of Observation":

It is the observed outcome due to the execution of a set of states of importance. For example, an exchange of messages is an observed outcome of certain operations in a system. Similarly, the failure of a communication mechanism is another event of observation.

Definition 5.2.3

"State Transition Graph" (STG):

The State Transition Graph is a state transition diagram where each state is a token state from a graph 'G' and the transitions between the states are exactly those corresponding to execution sequences of the token states defined by the set of elements of 'V' holding tokens simultaneously.

Definition 5.2.4

"Event State Transition Graph" (ESTG):

An "Event State Transition Graph" is the state transition graph corresponding to an event of observation.

There may be more than one event state transition graph corresponding to a given event of observation. The token states associated with different event state transition graphs are defined by different status transfer functions and the coordination constraints associated with the UGM representation of an event of observation.

The event state transition graph can be generated based on the event boundary conditions in the following systematic manner.

Algorithm 5.2

Step 1: Construct the SESX ["single entry single exit"] graph from a MEMX ["multiple entry" "multiple exit"] graph. (For definitions of SESX and MEMX graphs see [POST 74].)

Step 2: Initialization
Clear all tokens from all arcs in A.
Place a single token on S.

Step 3: Randomly select a node i.
Set $V = \{V - i\}$

Step 4: If all the incoming arcs into the node i possess tokens according to $\overleftarrow{M}$ take all the tokens Map $\overleftarrow{M}$ to $\overrightarrow{M}$ on i.

Step 5: If $c_0(i)$ belongs to $\overrightarrow{M}$ --> $S_0(i)$
then $S_0(i) = S_0(i) - C_0(i)$

Step 6: Place tokens on all $s_0(i)$ according to $\overrightarrow{M}$.

Step 7: Add $s_0(i)$ to the STG string.

Step 8: If v = 0 go to 3.

Step 9: Stop.

5.3 Application to Attribute Analysis

In order to analyse different attributes using the UGM, the system behavior is first represented by a UGM. A system's behavior is defined by identifying several major events pertaining to the system operations. The mapping function 'M' and the coordination constraints associated with these events of observation are defined. Once the major events of observations are defined, a tree is developed having observation event 'j' at the top of the tree and the basic events participating in the occurrence of this "observation event" (with the associated paramenters) as the branches of the tree.

For each major event of observation, an ESTG is first generated using Algorithm 5.2. Then, depending on the attribute and the conditions related to the attribute, a new state transition graph is generated. This new state transition graph is analysed using the proper termination property for appropriate characteristics. In the next section this issue is furthur discussed with respect to the reliability attribute, to show how this model can be used to model and analyse reconfigurability.

6. Study of Adaptive Reconfiguration

In this section the analysis of adaptive reconfiguration in a distributed environment is considered. The analysis procedure presented here considers several of the related attributes thus relating the abstract concepts of reconfiguration to the practical situations. There are three basic issues related to reconfigurability (i) conditions required for reconfigurability (ii) synthesis of reconfigurable strategies and

(iii) dynamic selection of a strategy.

6.1 Conditions for Reconfigurability

In order to establish the conditions for reconfigurability it is essential to identify the additional states and transitions introduced by the failures and see whether transitions from such states to normally valid states could be established. In general, a failure may lead to a sequence of new states and paths, these states and paths represent the several possible states and the associated arcs that may be created due to occurrence of a failure. Intuitively, the recovery and the failure reconfiguration involve, first the detection of the occurrence of such new states due to a failure and then the discovery of a path that would lead to a safe state in a finite number of steps. In addition, such a transition to safe states in a finite number of steps implies that such new states are finite in number and so is the number of transitions between them. These necessary conditions for the reconfigurability of a UGM in terms of the STG, ESTG (in this case the event is a failure) defined in section 6.2, can be stated as follows.

Necessary Conditions for Reconfigurability (NECOFOR)

(A) The number of failed states and critical states are finite.

(B) There is no failure F_1 for which the transition path of F_1 has a state transition having the same entity identification and boundary condition as the failure F_1.

(C) There exists a set of states and a transition sequence in ESTG whose occurrence can bring the system to a safe state.

However, in a distributed system it is not just sufficient to have these NECOFOR satisfied for the system to reconfigure and recover from a failure. First of all, the safe states that the new arcs would reach either from critical states or failed states may not be consistent in terms of the coordination constraints defining the overall operation of the system. Secondly, even if the individual coordination constraints are satisfied there might be additional restrictions from the performance point of view. So, it is essential to satisfy the coordination and timing constraints, in addition to the NECOFOR. These necessary and sufficient conditions can be formally stated as follows.

Theorem: A system represented by UGM is reconfigurable if and only if the NECOFOR is satisfied and the associated modified state transition graph is a consistent state transition graph.

The problem with the approach discussed so far is that in any real system, the enumeration of STG and ESTG is a stupendous task; hence we have to modify our analysis method.

In order to model and analyze a real system using the UGM approach, one could start with a known set of "observation events" instead of representing the entire system in a UGM representation. These known sets of events represent various major events with respect to different attributes. For example, while analyzing the reliability issue when adaptive failure reconfiguration is present in the system, one could start with a known set of failure events such as communication mechanism failure, a processor failure, memory system failure, I/O system failure, a process failure, and so on.

In order to study the feasibility of reconfiguration, failures at different levels of a hierarchical system are considered separately. A failure a higher level, is first considered and a corresponding ESTG is developed. This ESTG can be developed using different approaches discussed in the next section. Once an ESTG is developed in terms of the lower level failures and the associated states to the degree of resolution needed, a minimum set of basic events contributing to the original event of observation is found. Given these basic events, one could generate a failure behavior table describing the minimum set of basic events for each observation event of failure. This philosophy of analysis, based on the selection of certain types of observation events is consistent with the failure classification presented in Section 4, where the failures are classified based on the observed effect for uniform treatment of both software and hardware as well as design and operational failures. In order to define the major events of observation, we first describe the operation of the system with normal operating conditions, the I/O relations at a gross level, and the different attributes of importance. The next step is to define potential major failure events for the system and classify them as in Section 4.

On the basis of the information obtained on system operations and expected failure events, we derive the ESTG by first defining the failure transfer functions, which describe how the different subsystems or functions would contribute to the occurrence of various events of observation. From these transfer functions and system behaviour, there are several ways to generate the ESTG, the most powerful of which is Roth's 'D'-Algorithm which yields the basic minimum set of events contributing to a major event of failure. Using this set of events, the ESTG is generated.

To test for the reconfigurability of the system we use the conditions specified earlier in this section to see if the system can be recovered from each of the basic events. A system is reconfigurable if and only if the superimposed graph of basic events and the control graph corresponding to an observation event is properly terminated [POST 74].

6.2 Synthesis of Reconfiguration Strategy

The essential behavior that specifies a particular reconfiguration strategy to recover from a given failure is derived from the feasibility analysis of a reconfiguration. A system's behavior is first modelled as a UGM and the corresponding STG's are derived. Then the ESTGs corresponding to the failure of interest are derived. The new states and the associated transitions are also identified. Then a set of safe states to which a possible transition can be made from the failed or critical states are identified. Associated with these states and transitions are a set of resources and function calls that are required. These resources, operations, and features required for making these function calls, are selected and specified as described below.

6.2.1 Specification of Reconfiguration Strategy

A reconfiguration strategy contains functional modules for the failure detection and recovery together with the calls on reconfiguration function. A typical reconfiguration strategy is described below:

FAILURE DETECT < Process >

STATUS IN < Current status >

INCOORS < Coordination constraints >

FAILURE LOCATE < Location of failure >

SELECT < strategy > FROM < Strategy set >

NEED RESOURCE < units > OF < resource type >

REMOVE [all] < resource type > FROM < resource set >

ADD < resourse type > TO < resource set >.

CHANGE STATUS [all] < resource type > IN < needed resource set >

OUT STATUS < current state - next state >

OUT COORS < input coors > AND < Function - outcoors >

RESTART < process type > OUT OF < process list >

In this typical specification of the reconfiguration process, the invocation of FAILURE DETECT, FAILURE LOCATE, STATUS IN, INCOORS, will detect and locate the failures and collects the current status information and the coordination constraints respectively. Once the information is gathered, the SELECT and NEED RESOURCE, will evaluate this information and select a particular stratety, while NEED RESOURCE makes a request for the needed resources. Following a strategy selection, the constructs REMOVE, ADD, CHANGE STATUS, OUT STATUS, OUT COORS will selectively change the resource organization, and define the output conditions to be satisfied after the strategy implementation, in order to maintain the consistency of operation. The purpose of the OUT STATUS is to define the final status of various states at the end of execution of a reconfiguration call. Similarly, the OUT COORS define the output coordination states. After verifying this information for consistency of operation, certain processes are restarted by specifying through the construct RESTART.

6.2 Verification

The verification of a reconfiguration strategy involves the verfication of the consistency of operations after the reconfiguration is performed together with the verification of the control embedded with the specification of a reconfiguration strategy. This verification requires the establishment of formal correspondence between the model and the design of a reconfiguration strategy which can be achieved through the definition of functions, and the transitions required for reconfiguration i.e. the implementation of a reconfiguration strategy in terms of the structures and operations embedded into the system. Then a reconfiguration strategy is verified by constructing a transition table and checking for the consistency of input/output conditions associated with the transition table. The steps involved in the verification of a reconfiguration strategy can be summarized as follows.

Step 1: Define the input relations (input states, transitions and coordination constraints).

Step 2: Establish formal correspondence between the model and reconfiguration strategy.

Step 3: Construct a transition table for each reconfiguration strategy. Identify the reconfiguration functions needed, and the associated states and the coordination constraints.

Step 4: Establish a set of safe states whose members are implemented only by proper implementation of the reconfiguration strategy.

Step 5: For every reconfiguration strategy check whether the final states belong to the set derived in the step 4.

6.3 Location of Reconfiguration Points

6.3.1 Choice of a Reconfiguration Strategy

The adaptive reconfiguration approach provides an ultimate solution for highly reliable, continuously available computer system. However, in any real system, there may be several strategies available with varying degrees of complexity to recover from a failure. Hence the incorporation of several reconfiguration strategies built into the system and their choice during the system design and operation are needed to be supported by quantitative evaluation methods. Such an evaluation should indicate whether the designed and the selected strategies would meet the desired Availability, Survivability and Reliability requirements (RAS) and if so, their effectiveness.

In principle, the evaluation of any reconfiguration technique can be done by means of modelling using either analytical approach or experimental approach or a combination of both. The exact nature of the techniques depends on the class and the nature of failures. These failures may be permanent or transient in nature. In addition, there can be failures originating from the user during the design or operation phases. Hence any assessment of strategy should be able to cover all the aspects of the failures or at least should be able to distinguish the influence of various factors on the assessment strategies. In the literature [ARNO 75, AVIZ 75, BEAU 78, KONA 75] on evaluation models, several Figures of Merit, e.g. Mean Time to Failure, Mission Time Improvement Factor, Mean Time To Danger etc., have been used.

However, these figures of merit are based on certain assumptions and system configurations. First of all, these quantitative analyses require the knowledge of numerical failure rates which are assumed either to be constant or have certain type of distribution such as exponential. But seldom is the case in real systems. Also most of these evaluation strategies either ignore transient and partial failures or assume them to be permanent and no systematic techniques are available to verify and validate these assumptions. The cost concept is rarely considered as an evaluation parameter except in some cases where the figure of merit is defined interms of cost concept i.e. cost of failures and cost of reliability [HECH 73]. When redundancy and heterogeneity of different subsystems are considered the traditional quantitative evaluation strategies seem to be mathematically untrackable. In addition, some of these evaluation strategies assume 100% coverage which is rarely valid. Even in cases where safeguards have been designed for all physical failures some human operator errors may cause catastrophic failures form which there exists no auotmatic recovery.

4.3.2 Conceptual Model For Strategy Selection

Conceptually, the strategy assessment modelling can be represented as in figure 6.1. In this model for strategy assessment, 'S' represents safe state, 'R' represents the state in which the reconfiguration strategy would be initiated, and 'C' represents a set of critical states, while 'F' representing a set of states in which the system cannot function at an accepted level of performance. A system may enter these 'F' states either when the available redundancies are exhausted so that any further reorganization of resources is not possible, or when the systems performance of functioning subsystems is below the accepted level. A direct transition from a safe state to a failed state may also occur due to the occurrence of a failure for which there is no reconfiguration strategy built into the system (e.g. catastrophic human errors). This model takes a unified view of both permanent failures as well as transient failures. In addition, the design failures are also treated in the same uniform manner.

Representation of Transient, Permanent, Design and Operational Failures

A transient failure may damage the information of a system and may eventually lead to a permanent system failure. The reconfiguration after a transient failure requires the restoration of the damaged information so that the system can continue to function properly. In many cases, the recovery from these transient failures can be handled by reexecution of the computations performed during its presence. In terms of our model,

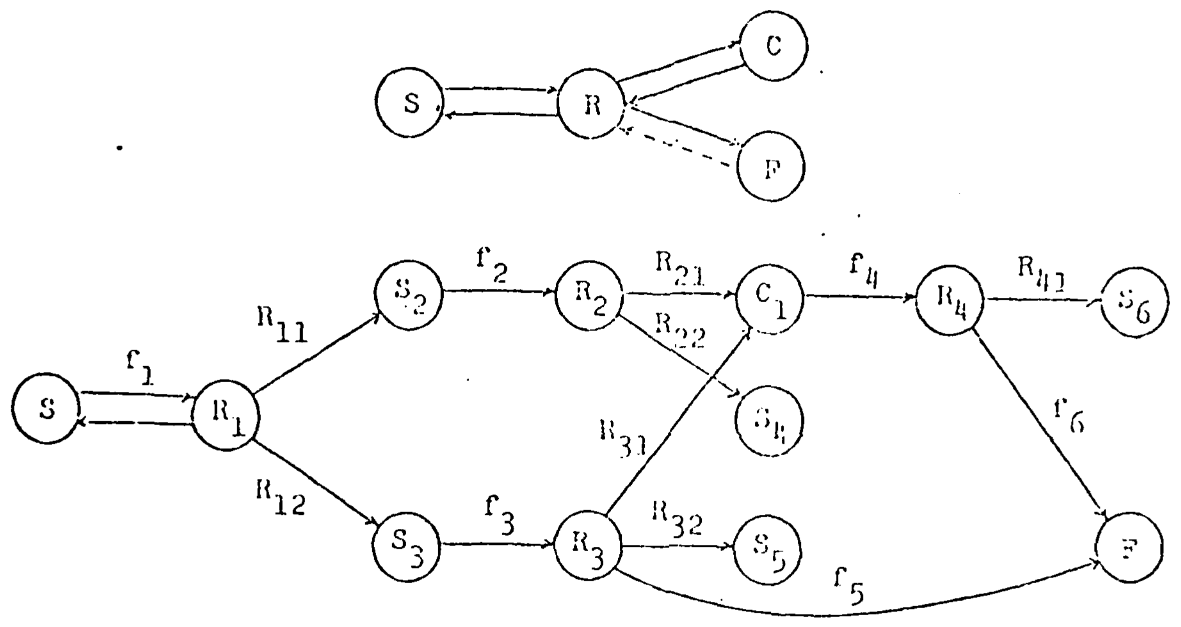


Fig. 6.1 - A Model for Strategy Selection

this is represented by a set of states and the associated transitions. This set of states represent the states after the occurrence of a failure, and the transitions are the transitions associated with the failures and the recovery actions. Failure of recovery may be due to the fact that a transient failure may be persistent or it may have damaged the vital information. In addition, it may also fail due to the inadequate resources for the recovery action or may be due dependent nature of some failures. In the model these cases are reflected by different paths taken by the recovery mechanism as shown in the modified diagram (figure 6.2).

The permanent failures are also represented by a set of new states and the associated transitions. They are characterized by the loss of processing modules. The recovery mechanisms for these failures usually involve some kind of redundancy, the higher the redundancy, the greater the chances of recovery.

In order to reflect the true effectiveness of a reconfiguration strategy in a distributed system, any measure such as a Figure Of Merit needs to consider all types of failures.

6.3.3 Derivation of a Figure Of Merit

Figure Of Merit (FOM) is a means of ordering various reconfiguration strategies for a given failure condition. It is also a means by which we compare two strategies. This FOM is useful throughout the design process to guide the choice of a strategy in designing a reconfigurable distributed system.

The Figures Of Merit for comparing different reconfiguration strategies are influenced by several factors. One of such factors is the importance of recovery from a particular failure in terms of the mission goal. Secondly,

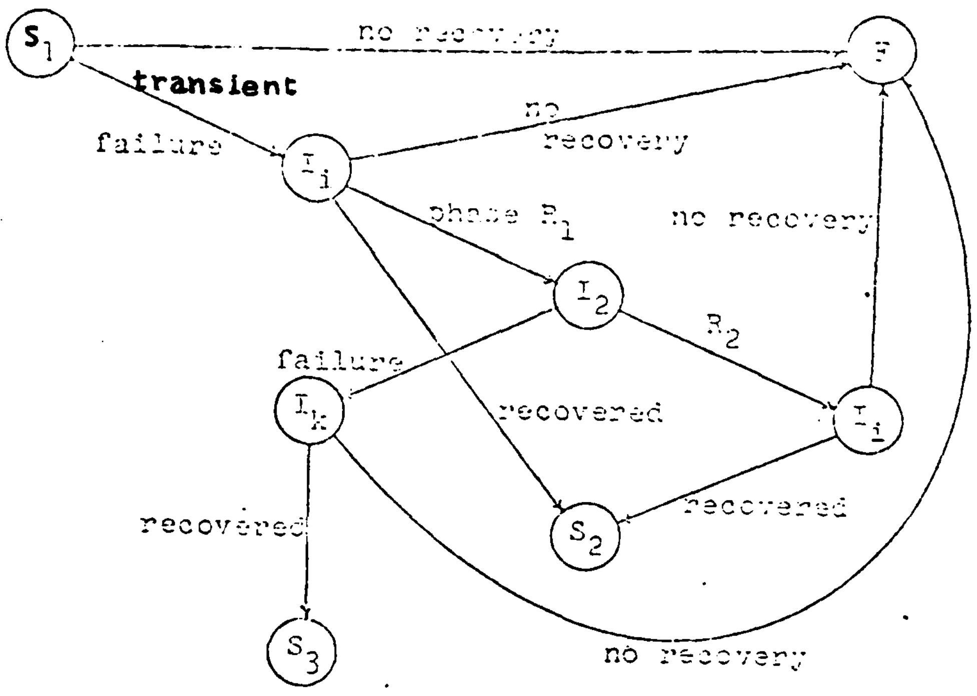


Fig. 6.2 - Modified State Diagram

one cannot assume that a reconfiguration is fault free and failures do not occur during its implementation, and such factors should be taken into account while deriving a Figure Of Merit. Also when more than one strategy is available to recover from a failure, any strategy may be selected with a finite probability. In addition, associated with every strategy that are required. Based on these factors a Figure Of Merit can be defined as follows:

A Figure Of Merit of a reconfiguration strategy $(FOM)_R$ is defined as

$$FOM_{Rij} = V_{Rij} F_{Rij} R_R \frac{1}{C_{ncf}}$$

where V_{Rij} if the vitality factor

F_{Rij} is the feasibility factor

R_R is the reliability of the strategy itself

C_{ncf} is the normalized cost factor

Vitality Factor

The vitality factor is defined as the rate at which the system availability increases as the probability of selecting a particular strategy increases. The vitality factor is a subjective estimate based on the designer's understanding of a recovery from a particular failure on resource allocation, performance, and mission goal. The factors that would influence the choice of a reconfiguration would also determine whether a reconfiguration is needed in the first place.

Feasibility Factor

The second factor involved in defining the Figure Of Merit is the feasibility factor associated with selection of a strategy. When more than one alternative is available the designer prefer one strategy to the other due to the nature and availability of the required resources. Even if a given strategy has higher vitality factor, higher reliability and low cost, it may not be selected bacause of lack of resources. The feasibility factor accounts for such situations.

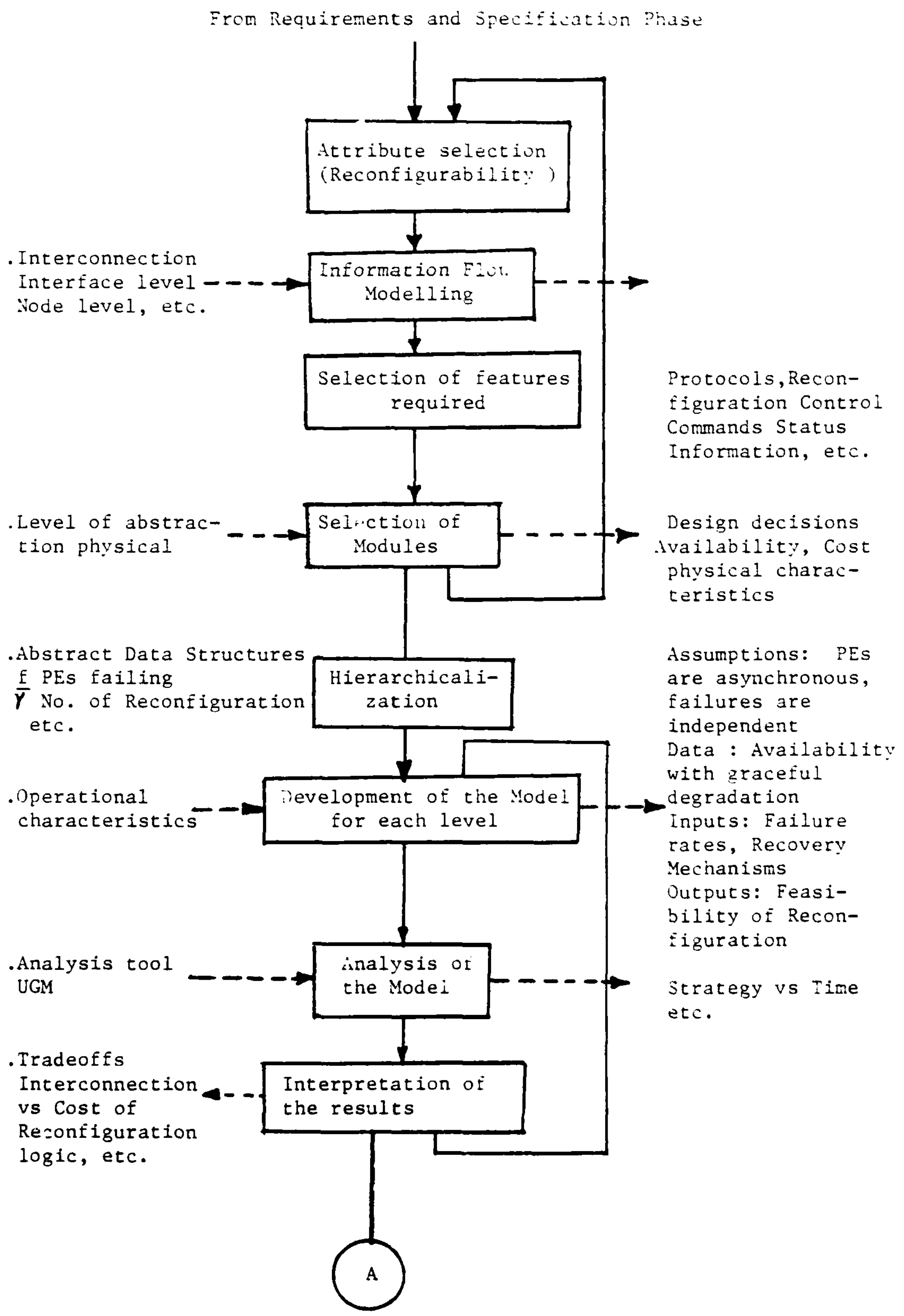
Reliability of a Strategy

The third important factor in determining the Figure Of Merit for a strategy is the reliability of the strategy itself. A failure may occur during recovery as during normal operation. Hence certain reconfiguration strategy may be more reliable than others, and this fact should be taken into consideration while deriving the Figure Of Merit. Then the reliability of a reconfiguration strategy (R_r) can be defined as the probability that the system would continue to function reliably after a strategy has been invoked.

Several techniques can be adopted to calculate the reliability of a path. One such technique is the approach used in fault tree analysis. As in the case of fault tree analysis, a tree is drawn with the event of reconfiguration as the top event and various steps or actions associated with the reconfiguration strategy as branches of the tree. Based on the reliabilities associated with the individual events, we could then calculate the reliability of the entire tree, i.e. reliability of the reconfiguration strategy itself.

Finally, the cost associated with the implementation of a strategy is another factor influencing the Figure Of Merit of a strategy. In certain cases, it may be worthwhile to contend with the suspiciously incorrect results than trying to recover from a failure. For example, in a batch processing or a time sharing system oriented towards nonreal time scientific computations, it may be economical to print an error message indicating that the results obtained are invalid since the detection of the last failure, rather than trying to recuperate from the failure by reconfiguration and program roll back because of the extensive overhead involved in reconstructing the status of the processes. Such similar considerations require the introduction of the cost parameter into the figure of merit. However, such a cost factor depends on the actual system conditions and the history of the failures and the recovery because the choice of current strategy is not only based on past failure history, but also determines the future failure and recovery sequences. Hence, the cost factor introduced should be normalized so that is is independent of the past history and the future consequences. Such a cost factor is the "Normalized Cost Factor" (C_{nef}). The normalized cost factor of a strategy is defined as the ratio of the weighted cost associated with that strategy to the cost associated with the reconfiguration adopted before that failure.

The calculation of a figure of merit provides a powerful tool in the design of a reconfigurable system. It provides the designer with vital information and guides him through the design process. Secondly, this methodology enables the designer to cover certain known failures thoroughly while considering them as an integral part of the system design approach to handle the associated complexity by shifting some of the dynamic decisions needed during operation zo the design phase. In addition, the associated knowledge is very useful in generating a strategy feasibility table for a given system and also in developing generalized provable strategies that only depend on the classes of failures and are independent of minute implementation details of a system.


From Requirements and Specification Phase
Attribute selection (Reconfigurability)
.Interconnection Interface level Node level, etc.
Information Flow Modelling
Selection of features required
Protocols, Reconfiguration Control Commands Status Information, etc.
.Level of abstraction physical
Selection of Modules
Design decisions Availability, Cost physical characteristics
.Abstract Data Structures
f PEs failing
γ No. of Reconfiguration etc.
Hierarchicalization
Assumptions: PEs are asynchronous, failures are independent
Data : Availability with graceful degradation
Inputs: Failure rates, Recovery Mechanisms
Outputs: Feasibility of Reconfiguration
.Operational characteristics
Development of the Model for each level
.Analysis tool UGM
Analysis of the Model
Strategy vs Time etc.
.Tradeoffs Interconnection vs Cost of Reconfiguration logic, etc.
Interpretation of the results
A

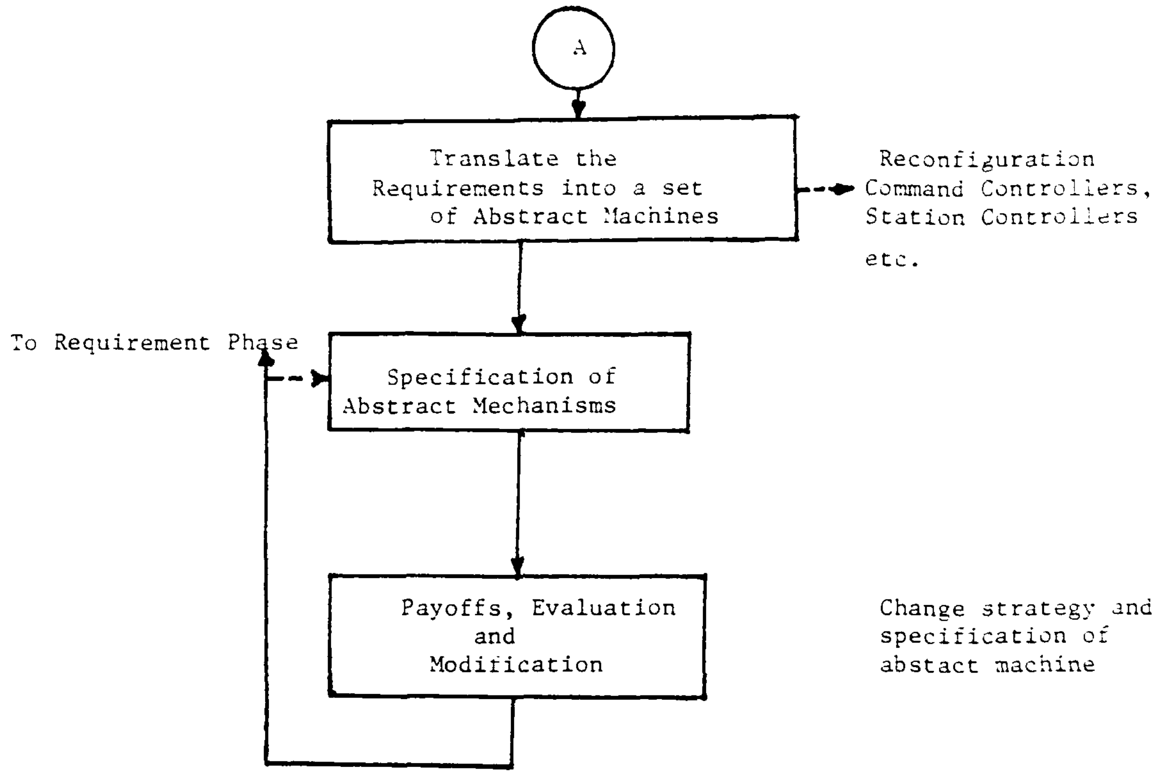

A
Translate the Requirements into a set of Abstract Machines
Reconfiguration Command Controllers, Station Controllers etc.
To Requirement Phase
Specification of Abstract Mechanisms
Payoffs, Evaluation and Modification
Change strategy and specification of abstact machine

Fig 7.1. Application of design methodology

7. Study of a Candidate Configuration

In this section we present an overview of a detailed design that was made of a candidate configuration based on the concepts summarized in earlier sections.

The configuration considered here takes the form of multiple processing elements connected via certain interconnection mechanisms.

7.1 Application of Design Methodology

The overall system requirements for the candidate configuration can be divided into data processing requirements, resource allocated, scheduling requirements and so on. Because of the distribution of processing functions among various processing nodes, one could expect that interprocessor communication will play a major role in system operation. Here we consider only the reliability and availability requirements and their influence on the architecture of the communication subsystem.

Figure 5.7 illustrates the application of the design methodology discussed in Section 5 to our problem.

7.2 Functional Requirements of the Communication System

The functions of the communication subsystem pertain to all the operations concerning the interprocessor communication and the associated failure detection and recovery operations. These include the interconnection mechanise, transmit function, receive functions, message switching dealing with message generation, checking etc., control functions dealing with synchronization, and changing the physical configxration, recovery functions dealing with failure detection and reconfiguration, and processing element dependent functions dealing with the memory access, interrupts etc. The UGM representation of these functions is given in

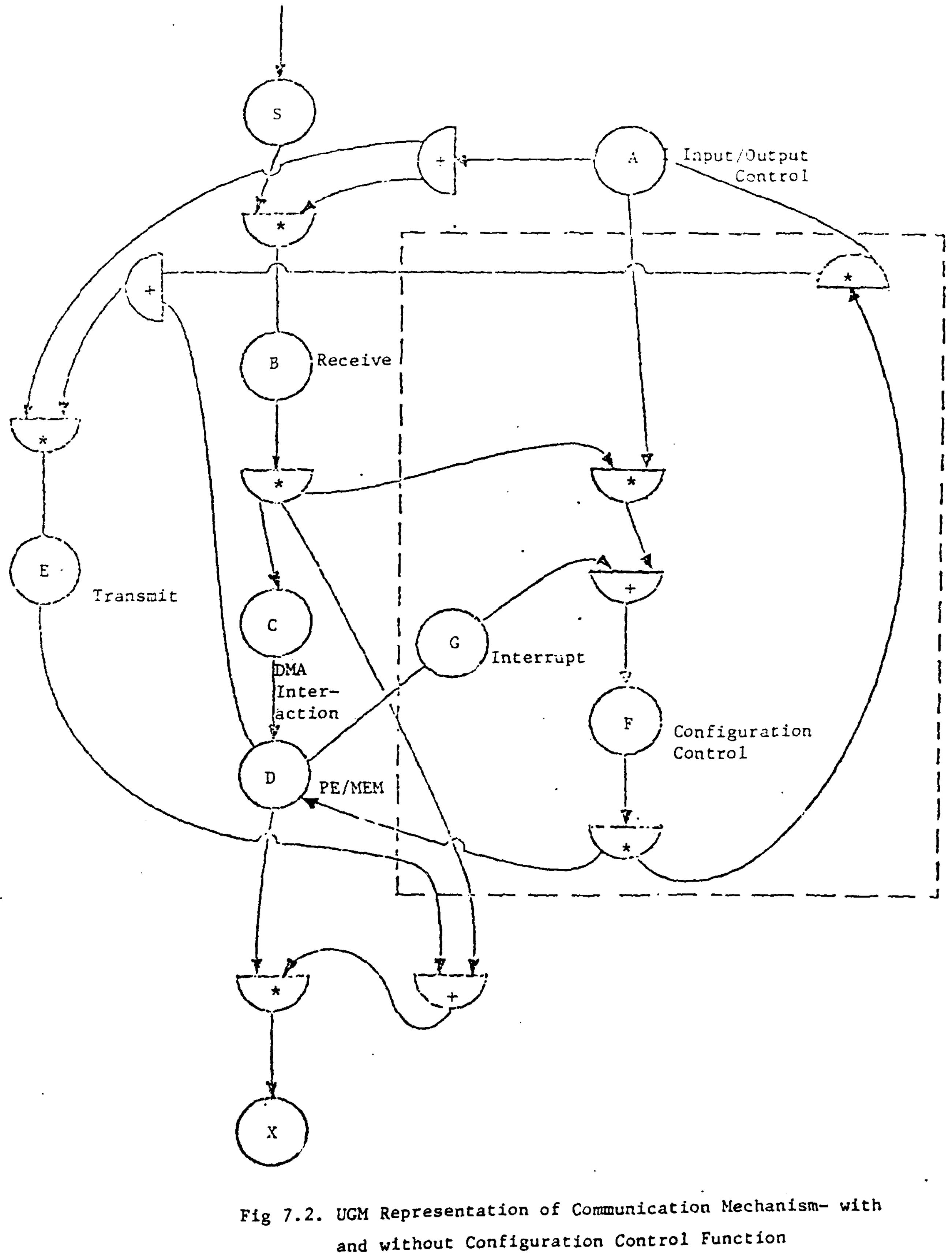
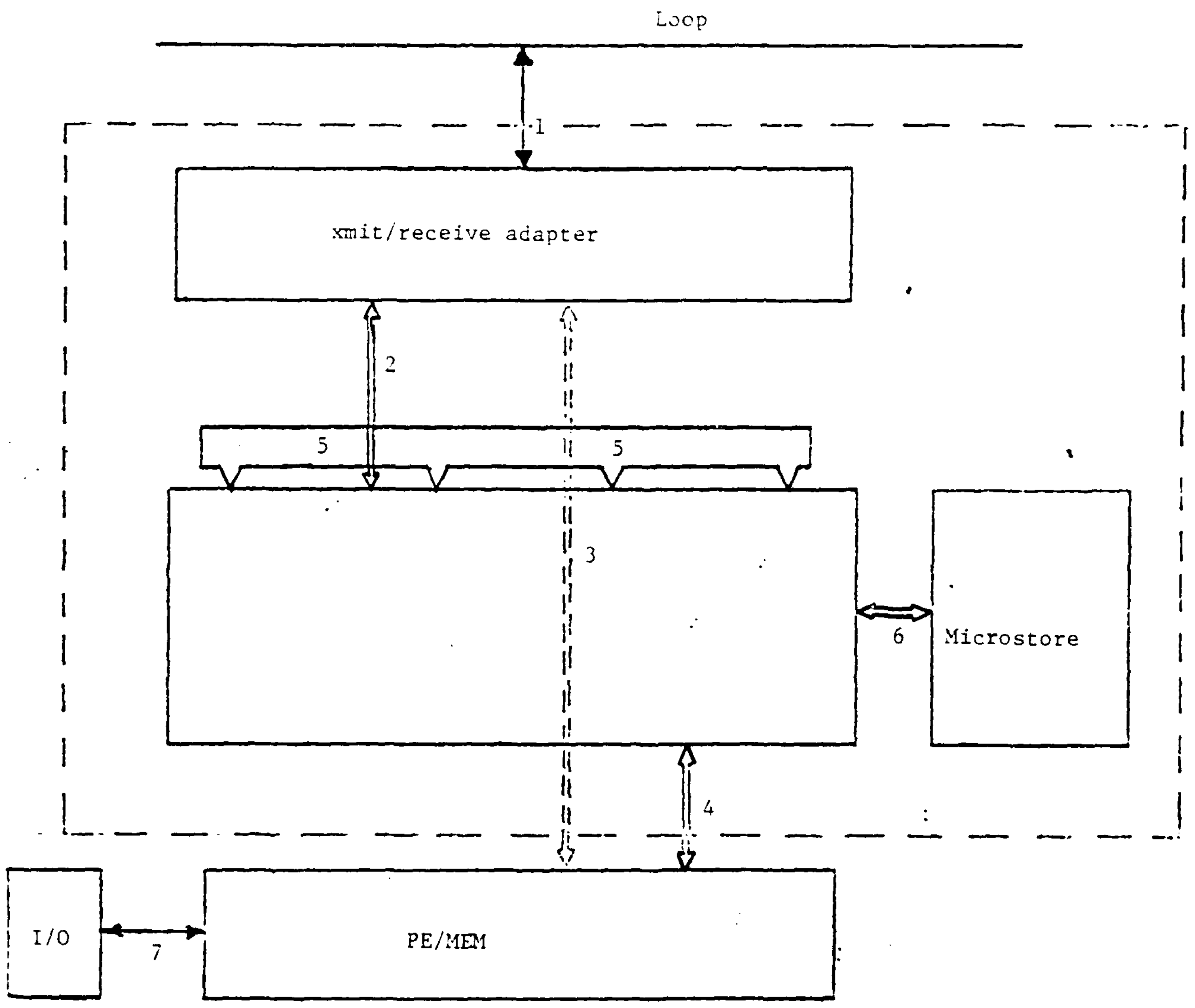


Fig 7.2. UGM Representation of Communication Mechanism- with and without Configuration Control Function



1. Signaling lines between CCC and Loop.
2. Remote Interface messages to and fro from interface functions.
3. Messages from PE to remote stations (application dependent).
4. Messages from CCC to local PE.
5. State Linkages between interface functions.
6. Microstore to in*terface functions.
7. Messages between I/O and PE or with secondary loop.

Fig.7.3. Data Paths of a CCC .

Fig. 7.2 and the associated data paths in Fig. 7.3.

7.3 The Interconnection Mechanism

There are several interconnection mechanism available such as single shared bus, system busses, crossbar switch, loop, mesh, etc. to interconnect the processing elements. [ANDE 75]. Of all these, the loop type of interconnection mechanism is chosen for our system because of its obvious advantages of simplicity, modularity and expandability, simple allocation and control structures, simple interface, and low cost. However, this interconnection mechanism has the limitations of low reliability, availability, and survivability, as the loop failures can be catastrophic. It also has limited bandwidth. The use of loop technology for data communication has been extensively studied and several modifications have been proposed to alleviate some of these limitations [YUEN 72, PIER 72, FARB 73, ZAFI 74, FRAS 74, REAM 75]. However, none of these methods can reconfigure for catastrophic failures with decentralized control.

The processing nodes are connected by means of the proposed loop interconnection mechanism via the interface units called Communication Coordination Controllers. The Communication Coordination Controller (CCC) connects various processing elements, and performs various communication functions related to the interprocessor communication. Functionally, a CCC consists of five functional modules i) Transmit/Receive unit (TR), ii) Input/output unit (IO), iii) Configuration Control unit (CC), iv) Message handler (MH), and v) PE/Memory dependent control unit (PM). The interprocessor communication is facilitated through a protocol defined in terms of certain system wide primitives.

7.4 UGM Representation of the Communication Coordination Controller (CCC)

The normal operation of the communication requires the functions related to transmission and reception, message handling, and functions related to system control and processing elements for its successful operation. Figure 7.2 is an UGM representation of the primitive operation of the communication subsystem when the failures requiring recovery occur. State "S" represents the starting of the operation. When the state "S" is executed as well as the proper signal from the input/output control state "A" is received, the DMA interaction state "C" and the PE/Memory state "D" are initiated. Once the message is received, the operation is terminated by executing the state "X". Similarly, the transmission is initiated when appropriate states are executed causing the execution of the state "E". Additional states and the transitions are introduced when the transmission and reception of the control information and the data are separated. The data paths between various functional units of CCC are shown in Figure 7.3. Now in order to incorporate the function of reconfiguration control and to change the system configuration in response to failures, additional states and transitions can be introduced (shown in dotted box).

7.5 Study of Failure Behavior and the System Design

In order to study the failure behavior of the communication subsystem it is first essential to identify several types of failures that are likely to occur in the example system. These failures may be due to the failure of the loop, or the interface unit (CCC), loss of communication primitives such as SYNC, SOM and related system wide control primitives, loss of communication functions, loss of data, parity errors, bit failures in crucial registers, failures in status words etc.

The role of a CCC in the failure behavior and recovery of the example system can be studied by first representing the functional behavior and identifying the failures associated with several functional units of a communication subsystem as discussed below.

Functions:
Establish electrical contact between all nodes via loop 1 (L1)
via loop 2 (L2)

Internal Events:
Loop 1 works (L1)
Loop 2 works (L2)
Loop 1 is open ($\overline{L1}$)
Loop 2 is open ($\overline{L2}$)

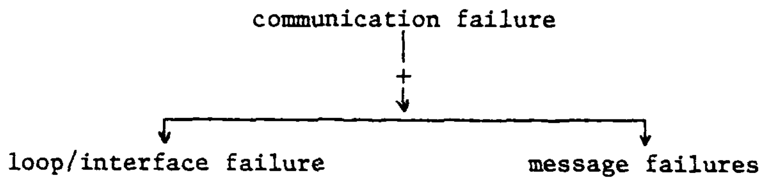
Transmit/receive control unit (TR)

Functions:
In bound clock extraction
Bit encoding and decoding
Loop checking
Detection of Sync/discard
Generate syne pattern
Detection of code violation
Nak generation
Parity generation/checking

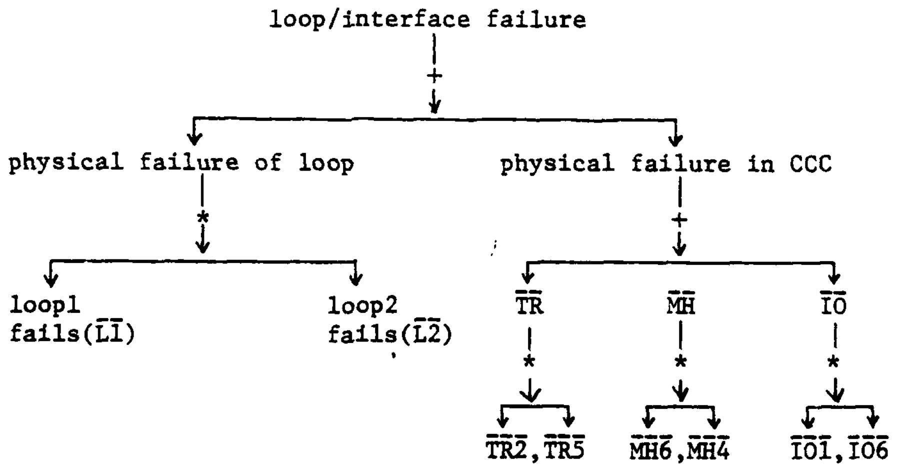
Internal Events:
In bound clock extraction ok (TRI)
In bound clock out of sync ($\overline{TRI}$)
Bit encoding and decoding ok (TR2)
Bit encoding and decoding error ($\overline{TR2}$)
Loop check ok (TR3)
Loop check not ok ($\overline{TR3}$)
Sync detector ok (TR4)
Sync detector fails ($\overline{TR4}$)
Sync generator ok (TR5)
Sync generator fails ($\overline{TR5}$)
Code violation detector ok (TR6)
Code violation detector fails ($\overline{TR6}$)
Ack/Nak generator ok (TR7)
Ack/Nak generator fails ($\overline{TR7}$)
Parity generator/detector ok (TR8)
Parity generator/detector fails ($\overline{TR8}$)

The module TR can cause failure of correct transmission of information due to its failure to perform any of its functions. The consequences of these failures may be erroneous transmission/reception of either data or control information. The failures associated with this module are identified by identifying the corresponding failure states and the transitions. Similar analyses are made for Input/Output Control (I/O), the Configuration Controller (CC), the Message Handler (MH) and the PE/Memory Controller (PM).

In order to develop recovery mechanisms to reconfigure the system from the identified failures it is essential to derive all the basic failure events contributing to the failure of successful communication. At the very top level unsuccessful communication may be due to either loop/interface (EEE) failure or message failures or both. This can be written as



The loop/interface failure, in turn, may arise from several causes:


loop/interface failure
+
physical failure of loop
physical failure in CCC
*
loop1 fails($\overline{L1}$)
loop2 fails($\overline{L2}$)
+
$\overline{TR}$
$\overline{MH}$
$\overline{IO}$
*
$\overline{TR2}$,$\overline{TR5}$
$\overline{MH6}$,$\overline{MH4}$
$\overline{IO1}$,$\overline{IO6}$

In generating the basic events causing an observed failure event, S and R refer to a sender and a receiver respectively. The bar (-) on any event refers to its failure mode. During normal operation both the loops as well as the physical functional units do not have any failures associated with them that can inhibit an electrical contact. Then the corresponding mincut set can be written as follows:

mincut set (during normal operation)
both loops work (L1 L2)
TR units of both sender and receiver work (TR_s TR_r)
MH units of both sender and receiver work (MH_s HM_r)
IO units of both sender and receiver work (IO_s IO_r)

There are 5 mincut sets when no electrical path exists, of which an example is

loops work (L1 + L2)
TR unit works (TR)
clock unit fails ($\overline{MH6}$)
buffer unit works (MH4)
sync generator works (IO1)

In order that the system can recover from the physical connectivity failure, it is necessary to recover from all the basic failure events defined by the mincut sets corresponding to a major connection failure.

Our proposal for the candidate configuration improves reliability by selecting a dual loop (Fig. 7.4 a,b) with "fold back". Normally, one of the loops is active. When a FOLD command is received by a CCC, it tries to transmit on both loops, while trying to receive on the active loop and monitoring the other. In order to perform these actions, several additional states and transitions must be introduced in the UGM representation.

STATES: configuration mode, sos mode, folding mode, switching mode, interface in, interface idle, system fail, power on 'xmit on loop 1, 'xmit on loop 2

TRANSITIONS: system fail, attention, fold, switch, monitor, loop choice, internal feedback, reset, interface enable, local PE on/off, unfold, sos signal

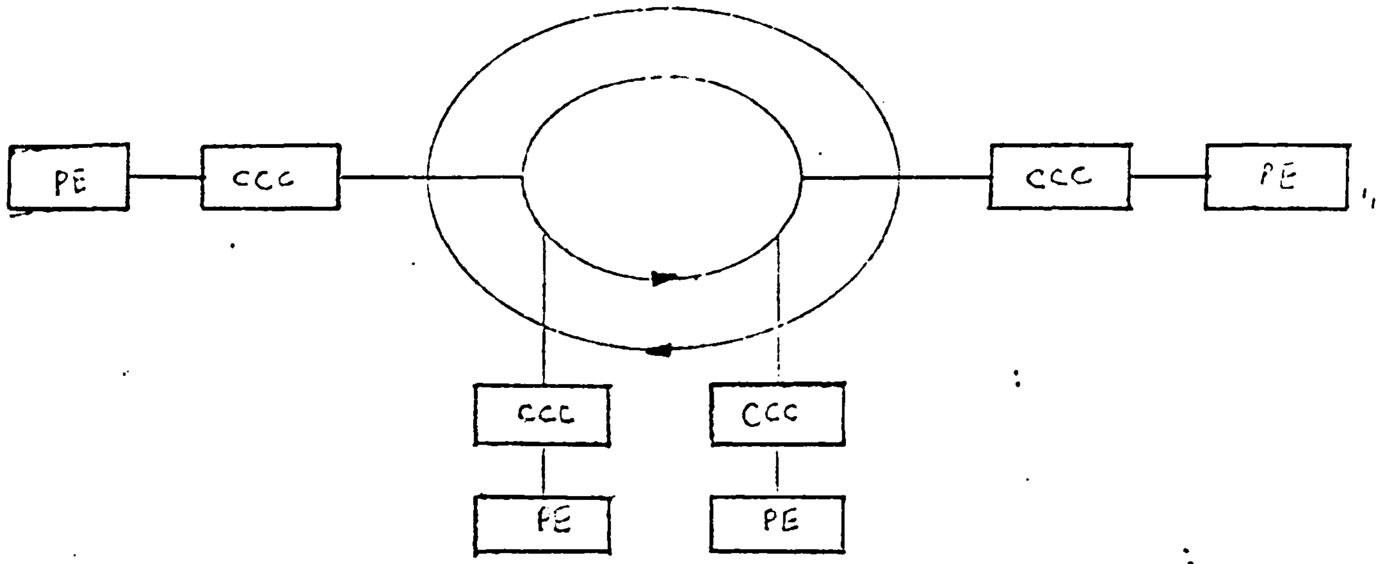


Fig. 7.4a System Architecture of the Proposed System

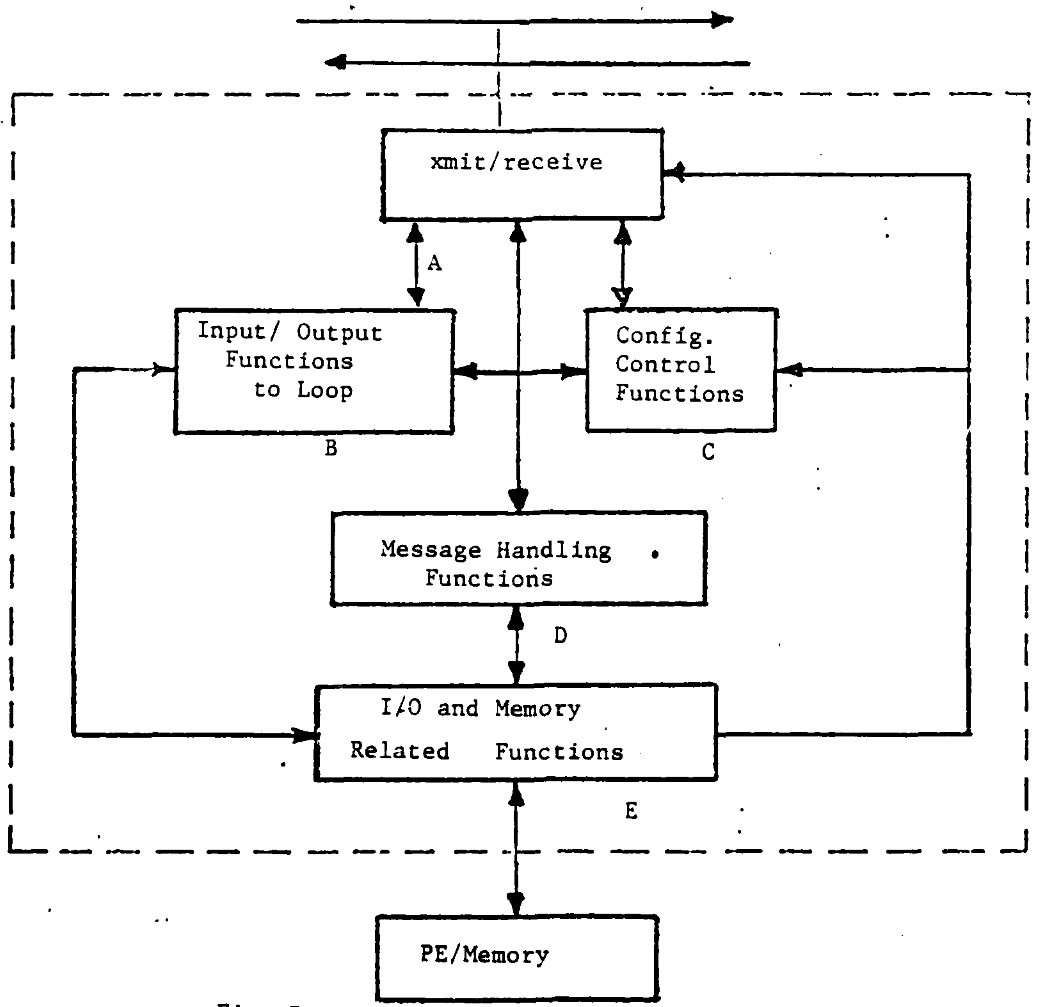

xmit/receive
A
Input/ Output Functions to Loop
B
Config. Control Functions
C
Message Handling Functions
D
I/O and Memory Related Functions
E
PE/Memory

Fig. 7.4b.The Structure of a CCC.

Based on the identified failures and transitions, reconfiguration strategies could be specified. The entire process is repeated for Message Transfer failures.

The next step is the identification of the states and transitions in the functional controllers associated with reconfiguration. For the Transmit and Receive Unit, as an example:

STATES: sender idle, sender generate, receiver idle, receiver interrupt, req. to send, clear to send, sync detect, receiver error, loop busy, sync generate, output word ready, input word ready

TRANSITIONS: interface valid, interface hold, local

In order to reconfigure the system for failures, it is essential to detect and locate the failures. When a failure has occured and the CCC receives a failure signal, the CCC stop the ongoing communication and initiate the failure location and recovery mechanisms. The CCC is designed to identify several different types of failures within the communication subsystem based on the particular type of failure, the appropriate failure location programs are initiated.

7.6. Evaluation of proposed design

The following definitions of events are used in the analysis.

Let n be the number of processing nodes connected by a dual loop

L be the number of loop segments and $L = n-1$

D_{ccc} is the fraction of the time that a CCC is non operational

d_{1s} is the fraction of the time that a loop segment is non operational

q is the prob. of a loop segment failure and $p+q = 1$

A_i be the availability of the node i

X,Y,Z be the probabilities that a node i, the nodes 1 to i, and i+1, to n are available respectively.

If we assume that all CCCs are functional then the communication link availability is dictated by the loop reliability. The connumication link is said to be operational if a communication path can be established from a given node to any other node in the system. It is given by

$$A_{(\text{single loop})} = p^L \quad (1)$$

$$A_{(\text{dual loop})} = 2p^L - p^{2L} \quad (2)$$

$$A_{(\text{with folding})} = 2p^L - p^{2L} + Lp^{2(i-1)}q^2 \quad (3)$$

However, in real systems the availability of the CCCs also dictate the availability of the over all communication subsystem.

A loop segemnt is operational with a probability $(1-D_{ls})$, and the CCCs are operational with a probability $(1-D_{ccc})$. Since we are assuming that the software for loop control and communication is operational, the probability that any given loop segment (i) is operational is given by

$$X = (1-D_{ls})(1-D_{ccc})^2 \quad (4)$$

probability that the loop segments between 1 to i nodes are available is given by

$$Y = (1-D_{ls})^{i-1}(1-D_{ccc})^i \quad (5)$$

probability that the loop segments between i+1 to n nodes are available is given by

$$Z = (1-D_{ls})^{L-i-1}(1-D_{ccc})^{L-i} \quad (6)$$

Then the probability that a node i of the system is operational is given by

$$A_i = Z\{YZ + Y(1-Z) + Z(1-Y)\} \quad (7)$$

The equation (7) yields the availability of a node in the example design. As can be seen from the equations, the expression for availability is quite complex. Nevertheless, it provides the designer with a means of evaluating the proposed design. A more refined analysis would require consideration of the feasibility and cost factors.

8. Summary

In this report, we have discussed a systematic method to evaluate and verify the performance of concurrent systems. The system to be studied is first modelled by a Petri net. Based on the Petri net model, the system is classified into either (1) a consistent system, or (2) an inconsistent system. A consistent system is further subclassified into: (i) a decision-free system; (ii) a safe persistent system; (iii) a general system. The system classification and the results are summarized in Figure 8.1. The performance of decision-free systems and safe persistent systems can be computed quite efficiently. In the case of general systems, we have proven that the verification of system performance is NP-complete. An approach for computing the upper and lower bounds of the performance of a conservative general system is proposed. However, the bounds produced may be loose. For a non-conservative general system, no good heuristics are known. Further research is needed.

In the area of system deadlocks, we have studied different analysis techniques for system deadlocks. In particular, we have concentrated our discussion on deadlocks caused by conflicts in mutual exclusive accesses to resources with the constraint that each resource type has only one member. This includes deadlocks in concurrent systems which use binary semaphores, critical regions and/or monitors as their synchronization mechanisms. A formal graph model (the request-possession graph) is developed to study deadlocks in these systems. Based on the model, the necessary and sufficient conditions for deadlocks are derived. It is found that determining the safety of a system is NP-complete. A deadlock detection procedure which can be executed in $O(2^k n^3)$ steps is developed, where k is the number of critical regions and/or monitors, and n is the number of references to these critical regions and/or monitors. Based on the procedure, a systematic approach for the construction of deadlock-free systems is developed. In that approach, critical regions and/or monitors are grouped into sets. The deadlock-free condition within each set is ensured by preanalyzing the system by the deadlock detection procedure. The deadlock-free condition among sets is guaranteed by imposing a linear ordering among the sets.

In distributed data bases, due to the inherent communication delay, it is not easy to obtain a consistent view of a system. We have developed three deadlock detection protocols (1) a two phase deadlock detection protocol; (2) a one phase deadlock detection protocol; (3) a hierarchical deadlock detection protocol. In the first protocol, two status reports have to be sent to a control node from each site before a deadlock can be determined. In the second protocol, only one status report from each site is required. However, more information has to be kept at each site and has to be sent to the control node each time. Based on the second protocol, a hierarchical protocol is developed. The network is divided into clusters. Status reports are preprocessed by the control node of each site before they are sent to its parent. In this way, the communication overhead of the deadlock detection procedure is reduced.

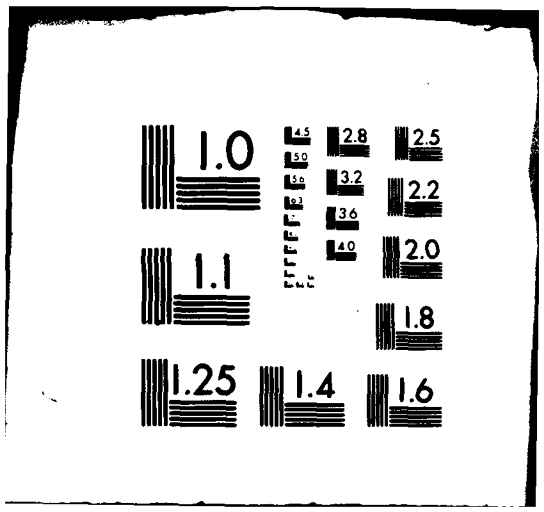
In the area of fault-tolerant distributed computer systems, we have tackled design problems related to reconfigurable distributed computer systems to increase their reliability, availability and survivability. We have investigated several concepts such as characterization of distributed systems and the associated failures, a systematic design approach and failure recovery via reconfigurability. Existing theories of failure detection and recovery fall short of their applicability to distributed computer systems as they failed to advance beyond the level of logic gates. In addition, these theories are developed with several idealistic assumptions which are seldom valid in the design of large scale systems. We have also developed an implementation independent failure classification in distributed computer systems. This classification which is based on the observed effects of failures rather than their causes would enable the system designer to deal with all possible types of hardware and software as well as design and operational failures. Moreover, various concepts such as feasibility of reconfiguration, and event of observation have been defined and discussed. A method of representing an event of observation in terms of the defined Unified Graph Model is described. The feasibility analysis then establishes a set of conditions that should be satisfied for a given system operation represented by a UGM, to recover from a failure. In addition, we have discussed the methods of specification and verification of reconfiguration strategies. A specified strategy is verified by verifying the control embedded into the specification of a strategy, and the consistency of operations after reconfiguration is performed. A quantitative measure (figure of merit) is also defined to compare and select a reconfiguration strategy.

Finally, we have demonstrated the applicability of various concepts

AD-A086 690 CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB F/6 9/2
THE DESIGN METHODOLOGY OF DISTRIBUTED COMPUTER SYSTEMS.(U)
MAY 80 C V RAMAMOORTHY AFOSR-78-3630
UNCLASSIFIED AFOSR-TR-80-0542 NL

2 of 2

END
DATE FILMED
8-80
DTIC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

and techniques developed through a detail design of an example distributed system. The goal of this system is to have built in reconfiguration features for failure recovery. First the proposed design methodology is applied to define detail architecture. In this system the reconfiguration is achieved by sending reconfiguration commands to various PEs through their CCCs. The CCCs themselves are self checking. Finally, failure behavior of the proposed system is studied and reconfiguration strategies for certain failures are derived.

However, this study on the design of reconfigurable distributed systems is based on certain assumptions and much remains to be done. One of the assumptions is that the designer has the knowledge of failure behavior of the system being designed. Secondly, this study is qualitative in nature and is based on subjective decisions based on the designer's understanding of a system being designed. More work needs to be done to incorporate the quantitative aspects. For example, the quantitative aspects can be incorporated on the study of strategy selection and on the effectiveness of reconfigurability. This report attempts to incorporate observability as a design parameter. More detailed study is needed to quantify and specify observability in terms of RAS requirements. The key factor for the success of a reconfigurable distributed computer system is the built in intelligence for self checking and recovery with distributed control. This is demonstrated through the design of the example system, the study of which is by no means complete and a more detailed study on its testability and feasibility of implementing CCC as an LSI element is warranted. It is also important to study effectiveness of various strategies on the communication. Last but not the least, the distributed operating system issues are not discussed explicitly, but requires a detailed study of these issues for the success of a truly reconfigurable distributed computer system.

Bibliography

[AHO 76] Aho, A. V., Hofcroft, J. E., Ullman, J. D., The Design and and Analysis of Computer Algorithms, Addison Wesley, 1976.

[AHU 79] Ahuja, V., "Algorithm to check Network States for Deadlock," IBM J. Res. Develop., Vol. 23, No. 1, January 1979.

[AGE 75] Agerwala, T. and Flynn, M. J., "On the Completeness of Representation Schemes for Concurrent Systems," Conference on Petri Nets and Related Method, M.I.T., Cambridge, Massachusetts, July 1975.

[ARM 76] R. G. Arnold and E. W. Page, "A Hierarchical Restructurable Multipurpose Mini Processor Architecture," Proc. 3rd Symp. on Computer Architecture 1976.

[AVI 71] A. Avizienis, "The STAR (Self Testing and Repairing) Computer: Investigation of the Theory and Practice of Fault Tolerant Computer Design," IEEE Trans. on Computers, November 1971.

[AVI 75] A. Avizienis, "Architecture of Fault-Tolerant Computing Systems," 1975 Int. Symp. on Fault-Tolerant Computing FTC-5, June 1975.

[BEA 78] M. D. Beaudry, "Performance Related Reliability Measures for Computing Systems," IEEE Trans. on Computers, June 1978.

[BOE 74] B. W. Boehm, "Some Steps Towards Formal and Automated Aids to Software Requirements Analysis and Design," IFIPS Proc., 1974.

[BRI 72] Brinch Hansen, P., "Structured Multiprogramming," Comm. ACM, Vol. 15, No. 7, July 1972.

[BRI 73a] Brinch Hansen, P., "Concurrent Programming Concepts," Computing Surveys, Vol. 5, No. 4, Dec. 1973.

[BRI 73b] Brinch Hansen, P., Operating System Principles, Prentice-Hall, Englewood Cliffs, N. J., 1973.

[COF 71] Coffman, E. G., Jr., Elphick, M. J. and Shoshani, A., "System Deadlocks," Computing Surveys, Vol. 3, No. 2, June 1971.

[COM 71] Commoner, F., et. al., "Marked Directed Graphs," J. of Computer and System Science, 5, 1971.

[DIJ 71] Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes," Acta Informatica, Vol. 1, No. 2, 1971.

[FLO 62] Floyd, R. W., "Algorithm 97, Shortest Path," Comm. ACM, 5 (1962) 345.

[FER 78] Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, Inc., Englewood Cliffs, 1978.

[GOS 71] Gostelow, K. P., "Flow of Control, Resource Allocation, and the Proper Termination of Programs," Ph.D. Dissertation, School of Engineering and Applied Science, University of California, Los Angeles, Dec. 1971.

[HAC 75] Hack, M., "Decidability questions for Petri nets," Ph.D. Thesis, Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., Dec. 1975.

[HAV 68] Havender, J. W., "Avoiding Deadlock in Multitasking Systems," IBM System J., 2, 1968.

[HECH 73] H. Heckt, "Figure of Merit for Fault Tolerant Space Computers," IEEE Trans. on Computers, March 1973.

[HOA 74] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," Comm. ACM, Vol. 17, No. 10, Oct. 1974.

[HOL 71] Holt, R. C., "On Deadlock in Computer Systems," Ph.D. Thesis, Dept. of Computer Science, Cornell University, Itaca, N. Y., Jan. 1971.

[KAR 66] Karp, R. M. and Miller, R. E., "Properties of a model for

Parallel Computation: determinancy, termination, queueing," SIAM J. Appl. Math. 14, 6, Nov. 1966.

[KAR 72] Karp, R. M., "Reducibility Among Comfinatorial Problems," Complexity of Computer Computations, Plenum Press, New York, 1972.

[KAR 77] Kartashev, S. I. and Kartashev, S. P., "Designing of LSI Modular Computers and Systems," MIMI 77, Proc. of Int. Symp. on Mini and Micro Computers, Montreal, November 1977.

[KON 78] Konakovsky, R., "Safety Evaluation of Hardware and Software," To be Presented at COMPSAC, November 1978.

[LIE 76] Lien, Y. E., "Termination properties of generalized Petri nets," SIAM J. Computer 5, 2, June 1976.

[LEH 76] Lehman, M. M. and Parr, F. N. "Program Evolution and its impact on Software Engineering," Proc. of the 2nd Int. Conf. in Software Engineering, Oct. 1976.

[MOR 74] Mortelmans, J., "An Investigation of a Parallel Reconfigurable Processor," Ph.D. dissertation Tech., report no. 3606-11, Stanford Electronics Laboratories, Stanford, March 1974.

[MUR 77] Murata, T., "Petri Nets, Marked Graphs, and Circuit System Theory," Circuits and Systems, Vol. 11, No. 3, June 1977.

[PAR 72] Parnas, D. L., "On the Criterias to be used in Decomposing Systems into Modules, Comm. of the ACM, Vol. 15, No. 12, Dec. 1972.

[PET 77] Peterson, J. L., "Petri nets," Computing Surveys, Vol. 9, No. 3, Sept. 1977.

[POS 74] Postel, J. B., "A graph Model Analysis of Computer Communication Protocols," Ph.D. Dissertation, Computer Science Dept., University of California, Los Angeles, Jan. 1974.

[RED 78] Reddi, S. S. and Feustel, E. A., "A Restructurable Computer System," IEEE Trans. on Computers, Jan. 1978.

[SCH 71] Schell, R. R., "Dynamic Reconfiguration in a Modular Computer System," Ph.D. dissertation M.I.T., 1971.

[ZAF 74] Zafiropulo, "Reliability - A key Element in Loop Systems," Int. Conf. on Digital Communications, Zurich, 1974.

DAT
FILM